Java 8 Ebook

Rahman Usta



Java 8 Ebook

İçindekiler

Java 8 Ebook

Rahman Usta

<rahmanusta@kodcu.com>

Önsöz

Java 8 ve yeniliklerini içeren bu kitap ile, Java programlama dilinin en yeni özelliklerini öğrenebilirsiniz.

Java 8 Ebook kodcu.com 'da Rahman Usta tarafından kaleme alınan Java 8 yazılarını içermektedir.

Bölüm 1. Lambda Expression nedir? Nasıl kullanırım?

Merhaba arkadaşlar;

Bugün sizlerle Java 8 (Project Lambda) ile Java ortamına katılan Lambda ifadelerinden (Lambda expression) bahsetmek istiyorum.

Lambda nedir?

Programlama dili çerçevesinde Lambda, anonim tekil görevler olarak değerlendirilebilir. Lambda deyimleri (Lambda fonksiyonları da denebilir), referans verilebilir ve tekrar tekrar kullanılabilirdir. Genel kullanım açısından Lambda fonksiyonları, diğer bir fonksiyona argüman olarak iletilebilirdir. Böylece, bir tarafta tanımlanan iş birimi, diğer bir iş biriminde koşturulabilir olmaktadır. Burada dikkat çekilecek unsur, bir iş biriminin diğer bir uygulama birimine referans olarak eriştirilebilirliğidir.

Javascript ile Lambda – Basit Giriş

Javascript dilinde `Array` sınıfının prototype alanında `forEach` isimli bir iş birimi (fonksiyon) bulunmaktadır. `forEach` fonksiyonu, parametre olarak bir anonim Javascript fonksiyonunu kabul etmektedir. `forEach` fonksiyonuna, parametre olarak tanımlanan anonim fonksiyon, `forEach` fonksiyonu içerisinde koşturulmaktadır. Bu sayede iş mantığı, diğer fonksiyona paslanmış olmaktadır.

Örnek 1:

```
var dizi = [1,2,3,4,5]; // Bir javascript dizisi

 // Anonim fonksiyon her bir elemanı çıktılıyor.
dizi.forEach(function(e){ //  (1)
    console.log("Eleman: ",e);
});
```

Yukarıda **(1)** numaralı kısımdaki anonim fonksiyon yani lambda fonksiyonu, forEach metodu içerisinde koşturulmaktadır. forEach metodu içerisindeki benzer yaklaşımı, kendi fonksiyonumuz ile oluşturalım.

```
var fonk= function(a,b,callback){
     // Lambda fonksiyonu "callback" burada koşturulur.
    return callback(a,b)*2; //  (2)
}

var result = fonk(2,3,function(a,b){ // (1)
     // Bu Lambda fonksiyonu 2 argümanı toplar.
    return a+b;
});

console.log("Sonuç ..: ",result); // Sonuç : (2+3)*2 = 10
```

Tekrar ederek, Lambda fonksiyonları için referans verilebilir iş birimleri denebilir. Bu yaklaşım bir çok programlama dilinde farklı biçimlerde yer alabilir, anlaşılabilirlik açısından ilk önce Javascript örneği vermeyi tercih ettim.

Lambda Java 8 öncesinde nasıldı?

Lambda birimleri (referans verilebilir iş birimleri), Java 8 evvelinde anonim sınıflar ile tanımlanabilmekteydi. Örnek 2'nin benzeri Java 8 evvelinde nasıl yazılıyordu?

Öncelikle bir iş birimi tipi gerekli, bu iş birimi 2 argüman kabul etmeli ve bir değer döndürmelidir. İş mantığı anonim olarak tanımlanacağı için, abstract sınıflar veya arayüzler kullanılabilir.

Örnek 3

```
public interface Anonim{
    public int call(int a, int b);
}
```

Anonim arayüz sözleşmesi, int türünden 2 değer alır ve bir int değer döndürür. Dikkat ederseniz sadece sözleşmenin maddesi tanımlandı, iş mantığı anonim olarak geliştirici tarafından tanımlanacak.

```
public class LambdaApp {

public static void main(String[] args) {

    LambdaApp app = new LambdaApp();

    // Örnek 2 (1) gibi
    app.fonk(2, 3, new Anonim() { ❶
        @Override
        public int call(int a, int b) {
            return a + b;
        }
     });

}

// Örnek 2 (2) gibi
public int fonk(int a, int b, Anonim anonim) {
    return anonim.call(a, b) * 2; ❷
};
```

```
}
```

❶ Anonim iş birimi burada tanımlanıyor

❷ Anonim iş birimi burada koşturuluyor

Java 8 ve Lambda ifadeleri

LambdaApp sınıfının (1) numaralı kısmındaki anonim iş birimi, Javascript dilindeki karşılığına göre şu anda çok daha kompleks. 6 komut satırı yer işgal ediyor. Fakat bu fonksiyonun yaptığı tek iş var, 2 argüman almak ve toplayıp geri döndürmek. Bu anonim birimi sanıyorum şu biçimde temsil edebiliriz.

```
fn(a,b) -> (a+b);
```

Görüldüğü üzere tek bir satırda aynı işi gören anonim iş birimini yani lambda ifadesini yazdık. Aynı örneği şimdi Java 8 Lambda deyimi ile yazalım.

```
public interface Anonim{
    public int call(int a, int b);
}

public class LambdaAppJava8 {

public int fonk(int a, int b, Anonim anonim) {
    return anonim.call(a, b) * 2;
};

public static void main(String[] args) {

    LambdaAppJava8 app = new LambdaAppJava8();

    app.fonk(2, 3, (a, b) -> a + b); // Dikkat (1)
}

}
```

Görüldüğü üzere 6 satırlık anonim sınıf kullanımını, tek satıra düşürmüş olduk.

Lambda ifadeleri, Java 8 versiyonunun en çarpıcı yeniliklerinden biri. (Bkz. Project Lambda).

Sizler de Java 8 kullanmak istiyorsanız https://jdk8.java.net/ adresinden sisteminize kurabilirsiniz.

Bol Javalı günler dilerim.

Bölüm 2. Lambda ifadeleri ve Fonksiyonel arayüzler

Merhaba arkadaşlar;

Bugün sizlerle fonksiyonel arayüzlerden (`Functional Interfaces`) ve Lambda ifadeleri (Lambda Expressions) ile olan ilişkisinden basetmek istiyorum.

Functional Interfaces

Tek bir soyut metodu bulunan arayüzlere fonksiyonel arayüz denir. İki veya üç değil, yalnızca bir tane soyut metodu olmalı. Peki neden `“1”` dersek sebebinin Lambda ifadelerine dayandığını söylemek isterim.

Örneğin;

Şöyle bir Lambda ifademiz olsun;

```
fn(x,y) -> 2*x + y
```

Bu Labmda deyimini örneğin Javascript gibi bir dil ile temsil etmek isteseydik şöyle yazabilirdik;

```
function(x , y) {

   return 2*x+y;

}
```

Peki fn(x,y) → 2*x + y Lambda ifadesini Java programlama dilinde nasıl yazabiliriz?

Java 8 geliştirilirken, geliştirim takımı kendilerine bu soruyu sormuş ve yanıt olarak fonksiyonel arayüzler kullanarak ihtiyacı karşılamışlar.

Şöyle ki; Bir Lambda ifadesi, Java derleyicisi tarafından, bir Java arayüzünün (tek bir soyut metodu olmalı) nesnel karşılığına (implementation) dönüştürülmektedir.

Örneğin, `fn(x,y) → 2*x + y` Lambda ifadesi, aşağıdaki fonksiyonel arayüze dönüştürülebilirdir. Çünkü fonksiyon x ve y adında iki parametre almakta ve `2*x + y` ile tek bir değer döndürmektedir. Tabi burada `x` ve `y` ' nin tipleri ne? dersek, herhangi bir matematiksel tip olabilir. Biz burada sabit bir tip (int gibi) verebilir, veya jenerik tiplemeler de yapabiliriz. Fakat şu anlık bununla yetinelim.

```
@FunctionalInterface // Opsiyonel
interface Foo{

    int apply(int x, int y);

}
```

Şimdi bu Lamda ifadesini uygulamamızda kullanalım.

Örnek 1:

Lambda ifadeleri atama sırasında kullanılabilir.

```
Foo foo = (x,y) -> (2*x+y);

int sonuc = foo.apply(3,4);

System.out.println("Sonuç: "+sonuc); // Çıktı: 10

// veya

Foo foo = (x, y) -> Math.pow(x,y);

int sonuc = foo.apply(3,2);

System.out.println("Sonuç: "+sonuc); // Çıktı: 3*3 = 9
```

Örnek 2:

Lambda ifadeleri, metod parametrelerinde de tanımlanabilir.

```
class Bar{

    public int calculate(Foo foo, int x, int y){
        return foo.apply(x,y);
    }
}

Bar bar = new Bar();

int sonuc = bar.calculate( (x,y) -> (2*x+y) , 3, 4 );

System.out.println("Sonuç: "+sonuc); // Çıktı: 10
```

Java programlama dilinde fonksiyon ifadesi pek kullanılmaz, onun yerine metod ifadesi kullanılır. Java metodları tek başına birer eleman değildir, diğer bir dil elemanının (sınıf, enum, interface ..) dahilinde tanımlanırlar. Javascript gibi dillerde ise fonksiyonlar tek başına birer elemandır, diğer bir dil elemanının içinde olmak zorunluluğu yoktur. Bu sebeple Java ortamında, bir Lambda ifadesinden bir fonksiyona/metoda/iş birimine dönüşüm için fonksiyonel arayüzler kullanılmaktadır.

Tekrar görüşmek dileğiyle..

Bölüm 3. Tarih ve Saat işlemleri

Java 8 içerisinde zaman temsili için yeni bir Date-Time API geliştirildi. Java programlama dilinde zamansal işlemler için JSR 310: Date and Time API şartnamesi yer almaktadır. Java 8 ile birlikte yeniden düzenlenen bu şartname, yepyeni özellikler sunmaktadır. Yeni geliştirilen Date-Time API'nin referans implementörü ThreeTen projesidir.

Yeni Date-Time API ile kolay kullanım, anlaşılırlık, thread-safety gibi hususlarda iyileştirmeler karşımıza çıkıyor. Bu yeni API' ye dair bileşenler (sınıf, arayüz, enum vs.) java.time paketi altında yer almaktadır. Şimdi sizlerle java.time paketi içerisinde yer alan bileşenler ile ilgili kolay anlaşılır örnekler paylaşmak istiyorum.

java.time paketindeki sınıfların genel olarak kurucu metodu private erişim belirleyicisine sahiptir, bu sebeple new anahtar ifadesiyle oluşturulamazlar. Onun yerine now, of , from, parse gibi metodlarla yeni nesneler oluşturulabilmektedir. java.time paketi içerisindeki zamansal sınıflar immutable' dir. Bu sebeple bir nesne oluşturulduktan sonra içerisindeki veriler kesinlikle düzenlenemezler. Bu da mevcut sınıfları thread-safety hale getirmektedir. (Bkz. Thread safety & immutability). Bu yazımızda `LocalDate`, `LocalTime`, `LocalDateTime`, `ZoneId` ve `ZonedDateTime` sınıflarının kullanımını irdeleyeceğiz.

LocalDate

LocalDate sınıfı ile tarihsel zaman temsil edilir. Örneğin: `10/10/2010` tarihini `LocalDate` ile temsil edebiliriz.

Örnekler

"of" metodu ile `LocalDate` nesnesi oluşturulabilir.

```
LocalDate.of(1988, 07, 16); // 1988-07-16
LocalDate.of(1988, Month.JULY, 16) // 1988-07-16 (Month enum)
```

"now" metodu ile o anın tarihi elde edilir.

```
LocalDate now = LocalDate.now(); // 2014-04-05
```

"with*" metodu ile eldeki bir `LocalDate` için gün, ay, yıl alanları düzenlenebilir. LocalDate sınıfı `immutable` 'dir. Bu sebeple with metodu farklı bir LocalDate nesne döndürür. O anki nesneyi düzenlemez.

```
LocalDate now = LocalDate.now(); // 2014-04-05

now.withYear(2016); // 2016-04-05

now.withMonth(8).withDayOfMonth(17); // 2014-08-17

now
.with(ChronoField.YEAR, 2012)
.with(ChronoField.MONTH_OF_YEAR, 8)
.with(ChronoField.DAY_OF_MONTH,3); // 2012-08-03
```

"plus*" metodu ile eldeki bir LocalDate için gün, ay, yıl alanları artırılabilir. `LocalDate` sınıfı immutable'dir. Bu sebeple `plus` metodu farklı bir LocalDate nesne döndürür. O anki nesneyi düzenlemez.

```
now
.plusWeeks(2)
.plusDays(3)
.plusYears(3)
.plusDays(7); //

now
.plus(2, ChronoUnit.WEEKS)
.plus(3, ChronoUnit.YEARS)
.plus(3, ChronoUnit.DECADES); //
```

"minus*" metodu ile eldeki bir LocalDate için `gün`, `ay`, `yıl` alanları azaltılabilir. `LocalDate` sınıfı immutable'dir. Bu sebeple minus metodu farklı bir LocalDate nesne döndürür. O anki nesneyi düzenlemez.

```
now.minusDays(5).minusMonths(2); //

now
.minus(2, ChronoUnit.WEEKS)
.minus(3, ChronoUnit.YEARS)
.minus(33, ChronoUnit.DAYS); //
```

LocalTime

LocalTime sınıfı ile saatsel zaman temsil edilir. Örneğin: `20:11` , `18:15:55` saatlerini LocalTime ile temsil edebiliriz. `LocalTime` ile saat, dakika, saniye, salise temsil edilebilir.

Örnekler

```
LocalTime now= LocalTime.now();

LocalTime time= LocalTime.of(18, 20, 55); // 18:20:55

time.getHour(); // 18
time.getMinute(); // 20
time.getSecond(); // 55

LocalTime.NOON; // 12:00
LocalTime.MIDNIGHT; // 00:00

LocalTime.parse("10:05"); // 10:05

time
.plusSeconds(45)
.minusSeconds(5)
.minus(5, ChronoUnit.SECONDS); // 18:20:35
```

LocalDateTime

LocalDateTime sınıfı ile hem tarihsel hem saatsel zaman temsil edilir. Örneğin: `10/10/2010 15:22:33` zamanını `LocalDateTime` ile sınıfsal olarak temsil edebiliriz.

Örnekler

```
LocalDateTime now = LocalDateTime.now();

LocalDateTime dateTime =
              LocalDateTime.of(2014, Month.JANUARY, 3, 4, 5,
45);

dateTime.getYear(); // 2014
dateTime.getMonth(); // JANUARY
dateTime.getDayOfMonth(); // 3
dateTime.getHour(); // 4
dateTime.getMinute(); // 5
dateTime.getSecond(); // 45

// LocalDateTime2LocalDate ve LocalDateTime2LocalTime
LocalDate date = dateTime.toLocalDate();
LocalTime time = dateTime.toLocalTime();

dateTime
.minusDays(3)
.plusYears(3)
.plusMinutes(3)
.minusWeeks(5)
.plusSeconds(2)
.plus(3, ChronoUnit.DECADES)
.minus(3, ChronoUnit.DECADES);
```

ZoneId

ZoneId sınıfı, dünya üzerindeki saat dilimlerinin Java taraflı nesnel karşılığını temsil için oluşturulan yeni bir Java 8 bileşenidir.

Java 8 için tüm saat dilimlerinin listelenmesi için `ZoneId` sınıfının `getAvailableZoneIds` metodu kullanılabilmektedir. Örneğin aşağıdaki kod, tüm saat dilimlerini sıralı olarak listelemektedir.

```
Set<String> zones = ZoneId.getAvailableZoneIds();

zones.stream().sorted().forEach(System.out::println);
```

```
…
Africa/Abidjan
Africa/Accra
Africa/Addis_Ababa
…
America/Argentina/Tucuman
America/Argentina/Ushuaia
America/Aruba
America/Asuncion
…
Asia/Istanbul
Asia/Jakarta
Asia/Jayapura
…
Etc/GMT+12
Etc/GMT+2
Etc/GMT+3
…
Europe/Isle_of_Man
Europe/Istanbul
Europe/Jersey
…
```

O anki çalışılan makinadaki saat dilimi için ZoneId sınıfının systemDefault metodu kullanılabilir.

```
ZoneId.systemDefault(); // Europe/Athens
```

JVM varsayılan saat dilimini mevcut işletim sistemi üzerinden almaktadır. Eğer istenirse mevcut Java sanal makinesinin varsayılan saat dilimi `-Duser.timezone=<Time Zone>` ifadesiyle düzenlenebilmektedir.

`-Duser.timezone=Europe/Istanbul` gibi.

ZonedDateTime

ZonedDateTime sınıfı aslında saat dilimi katıştırılmış LocalDateTime sınıfıdır desek yeridir. LocalDateTime sınıfından farkı genel olarak temsil ettiği zamanı saat dilimi içerir olarak sunmasıdır.

Örnekler

```
ZonedDateTime.now();
/* 2014-04-05T16:33:16.320+03:00[Europe/Athens] */

ZoneId istanbul = ZoneId.of("Europe/Istanbul");
ZonedDateTime.now(istanbul);
// 2014-04-05T16:33:16.330+03:00[Europe/Istanbul]

// Japonyada tarih/saat kaç?
ZonedDateTime.now(ZoneId.of("Japan"));
// 2014-04-05T22:33:16.331+09:00[Japan]
```

Bir `ZonedDateTime` nesnesi `LocalDateTime`, `LocalDate` ve `LocalTime` karşılıklarına dönüştürülebilmektedir.

```
LocalDateTime localDateTime = japan.toLocalDateTime();
/* 2014-04-05T22:33:16.331 */

LocalDate localDate = japan.toLocalDate(); // 2014-04-05
LocalTime localTime = japan.toLocalTime(); // 22:33:16.331
```

Kaynaklar

[1] http://www.threeten.org/ [2] http://java.dzone.com/articles/introducing-new-date-and-time [3] http://docs.oracle.com/javase/tutorial/datetime/index.html

Tekrar görüşmek dileğiyle..

Bölüm 4. Consumer Arayüzü

Daha önceki yazılarımızda Lambda ifadelerinden ve Fonskiyonel arayüzlerden bahsetmiştik. Şimdi ise, `java.util.function` paketi altında yer alan ve gömülü olarak bulunan fonksiyonel arayüzlere değineceğiz.

java.util.function paketi altında, farklı amaçlar için bulunan hazır arayüzler bulunmaktadır. Java 8 içerisinde Lambda deyimlerinin kullanılabilir kılınmasında, java.util.function paketi altındaki arayüzler kullanılmaktadır. Bu fonksiyonel arayüzlere java.util.function adresinden görebilirsiniz.

Bu yazımızda ise, fonksiyonel arayüzlere giriş düşüncesiyle `java.util.function.Consumer` arayüzünden ve nasıl kullanıldığından bahsedeceğiz. `Consumer` arayüzü accept isimli tek bir metoda sahiptir. Bu fonksiyonun bulunuş amacı, tüketim operasyonlarında kullanılmasıdır. Tüketimden kasıt edilen ise, metoda girdi olması fakat çıktı olmamasıdır. Metod girdisinin tipi ise jenerik olarak `T` harfi ile temsil edilmiştir. `T` yerine, isteğe göre herhangi bir Java tipi gelebilir. Biz String tipini kullanacağız.

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t);

}
```

Consumer arayüzü, yukarıda görüldüğü üzere bir fonksiyonel arayüzdür. Bir arayüzün, fonksiyonel olarak nitelenebilmesi için, tek bir soyut metoda sahip olma şartı vardır. Zaten bu sayede, fonksiyonel arayüzler Lambda deyimlerine öykünebilmektedirler.

Java 8 öncesi

Java 8 öncesine göre Consumer arayüzü türünden bir nesneyi anonim bir sınıf ile oluşturulım.

```
Consumer<String> consumer = new Consumer<String>() {
   @Override
   public void accept(String msg) {
       System.out.println(msg);
   }
};

consumer.accept("Merhaba Dünya");
```

Görüldüğü üzere bir anonim sınıf oluşturduk ve accept metodunu tükettik.

Java 8 sonrası

Java 8 sonrası için anonim fonksiyonlar yerine Lambda deyimlerini kullanabilmekteyiz. Örneğin yukarıdaki anonim sınıfı aşağıdaki yer alan Lambda deyimi ile yer değiştirebiliriz.

```
Consumer<String> consumer = (String msg) -> {
    System.out.println(msg);
};

consumer.accept("Merhaba Dünya");
```

Lambda deyimlerinde odaklanması gereken nokta, fonksiyonel arayüzün tek metodunun sahip olduğu metod girdi tipi, sayısı,sırası ve metod çıktı tipidir. Bu sayede kod satırı olarak tasaruf edilmektedir.

Lambda deyimleri akıllıdır

Lambda fonksiyonlarında tanımlanan metod girdi tanımlamalarında, istenirse tip tanımlamasından feragat edilebilir. Yani yukarıdaki Lambda deyimini aşağıdaki gibi yazabiliriz.

```
Consumer<String> consumer = (msg) -> {
    System.out.println(msg);
};

consumer.accept("Merhaba Dünya");
```

Görüldüğü gibi, (String msg) tanımlamasını (msg) yapabildik. Sol tarafta zaten String tip bilgisi yer aldığından, compiler metod girdisinin tipini buradan elde edecektir. Tip tanımlamalarından feragat ettiğimiz gibi parantezden de kurtulabiliriz.

```
Consumer<String> consumer = e -> {
    System.out.println(e);
};

consumer.accept("Merhaba Uranüs");
```

Consumer arayüzü nerede kullanılıyor?

JDK 8 in çekirdeğinde java.util.function arayüzleri halihazırda kullanılmaktadır. Örneğin Iterable arayüzünün içerisinde forEach metodunda Consumer arayüzü kullanılmaktadır.

```
public interface Iterable<T> {

    ...

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }

    ...
}
```

forEach metodu, önce null kontrolü yapmakta ve ardından, döngüsel olarak mevcut veri tipi üzerinde veri tüketimi yapmaktadır. Şimdi bu metodu kullanan basit bir örnek yazalım.

```
List<String> names = Arrays.asList("Ali", "Veli", "Selami");

names.forEach(consumer);

// veya

names.forEach(e -> {
   System.out.println(e);
});
```

Bu örnek içerisinde her bir isim bilgisi tek tek konsol ekranına çıktılanmaktadır.

Lambda deyimlerini Metod Referansları ile kullanabiliriz.

Java 8 evvelinde bir metodu referans olarak kullanma şansı bulunmuyordu. Fakat Java 8 ile birlikte Java metodlarını referans olarak kullanabiliyoruz. Örneğin, elimizde halihazırda aşağıdaki gibi bir listele metodu bulunsun.

```
public class App{

 public static void listele(String e) {
     System.out.println(e);
 }

}
```

Dikkat edilirse bu metodun girdi ve çıktı normu, Consumer#accept metodunun girdi ve çıktı biçimiyle birebir aynıdır. Bu sebeple, listele metodu metod referansı olarak Consumer tipi için kullanılabilirdir.

```
Consumer<String> consumer= App::listele;

consumer.accept("Merhaba Dünya");
```

Bir metodu referans olarak kullanabilmek için
ifadesi kullanılmaktadır. Şimdi metod referans kullanımını örnek olarak forEach metodu üzerinde deneyelim.

```
List<String> names = Arrays.asList("Ali", "Veli", "Selami");

names.forEach(App::listele);
```

`App#listele` metodunun bir benzerinin yazılmışı zaten PrintStream sınıfı içerisinde var.

```
List<String> names = Arrays.asList("Ali", "Veli", "Selami");

names.forEach(System.out::print);

// veya
```

```
names.forEach(System.out::println);
```

Şimdilik bu kadar, tekrar görüşmek dileğiyle.

Bölüm 5. Lambda örnekleri

`java.util.function` paketi altında bir çok fonksiyonel arayüz bulunmaktadır. Bu arayüzlerin temel amacı, farklı tipteki Lambda ifadelerine temel oluşturmaktır.

Consumer Arayüzü

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t); // t-> {}

}
```

T tipindeki parametreyi alır ve tüketir/işler. Geriye değer döndürmez (void). T burada herhangi bir sınıf tipi olabilir.

Consumer Arayüzü Örnek.

```
Consumer<String> consumer = word -> {
            System.out.println(word); // Merhaba Dünya
        };

consumer.accept("Merhaba Dünya");
```

BiConsumer Arayüzü

```
@FunctionalInterface
public interface BiConsumer<T, U> {

    void accept(T t, U u); // (t,u) -> {}
}
```

T ve U tiplerinde iki parametre alır ve bu parametreleri tüketir. Geriye değer döndürmez.

BiConsumer Arayüzü Örnek.

```
BiConsumer<String, Integer> biConsumer = (name, age) -> {
            System.out.println(name+":"+age); // Alinin yaşı:25
        };
biConsumer.accept("Ali'nin yaşı",25);
```

Function Arayüzü

```
@FunctionalInterface
public interface Function<T, R> {

    R apply(T t); // t-> r

}
```

T tipinde bir parametre alır, işler ve R tipinde bir değer döndürür.

Function Arayüzü Örnek.

```
Function<Integer, Integer> function = t -> Math.pow(t,2);
Integer result = function.apply(5);
System.out.println(result); // 25
```

UnaryOperator Arayüzü

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {

}
```

Function türündendir. Eğer T ve R tipleri aynı türden ise, ismi UnaryOperator olur.

UnaryOperator Arayüzü Örnek.

```
UnaryOperator<Integer> unaryOperator = a -> Math.pow(a,5);
Integer result = unaryOperator.apply(2);
System.out.println(result); // 32
```

BiFunction Arayüzü

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    R apply(T t, U u); // (t,u) -> r
}
```

T ve U tiplerinde iki parametre alır, R tipinde değer döndürür. T, U ve R herhangi bir sınıf tipi olabilir. Function#apply tek parametre alırken Bi* iki parametre alır.

BiFunction Arayüzü Örnek.

```
BiFunction<Integer, Integer, String> biFunction = (a, b) ->
"Sonuç:" + (a + b);
String result = biFunction.apply(3,5);
System.out.println(result); // Sonuç: 8
```

BinaryOperator Arayüzü

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T> {

}
```

BiFunction türündendir. T, U ve R aynı tipte ise BinaryOperator kullanılabilir.

BinaryOperator Arayüzü Örnek.

```
BinaryOperator<Integer> binaryOperator = (a, b) -> a + b;
Integer result = binaryOperator.apply(3,5);
System.out.println(result); // 8
```

Predicate Arayüzü

```
@FunctionalInterface
public interface Predicate<T> {

    boolean test(T t); // t-> true/false
}
```

T tipimde bir parametre alır, şarta bağlı olarak true/false değer döndürür.

Predicate Arayüzü Örnek.

```
Predicate<Integer> predicate = a -> (a > 0);

boolean pos = predicate.test(5); // true
boolean neg = predicate.test(-5); // false
```

BiPredicate Arayüzü

```
@FunctionalInterface
public interface BiPredicate<T, U> {

    boolean test(T t, U u); // (t,u) -> true/false
}
```

T ve U tiplerinde iki parametre alır, şarta bağlı olarak true/false döndürür.

BiPredicate Arayüzü Örnek.

```
BiPredicate<Integer, Integer> biPredicate = (a, b) -> (a > b);

boolean bigger = biPredicate.test(5,4); // true
boolean lower = biPredicate.test(5,7); // false
```

Supplier Arayüzü

```
@FunctionalInterface
public interface Supplier<T> {

    T get(); // () -> t
}
```

Hiç parametre almaz, T tipinde bir değer döndürür. Factory pattern için uygundur.

Supplier Arayüzü Örnek.

```
Supplier<List> supplier = () -> new ArrayList<>();
List<String> liste = supplier.get();
liste.add("Ali");
liste.add("Veli");
liste.add("Selami");
```

Tekrar görüşmek dileğiyle…

Bölüm 6. Notasyonlar ve Tekrarlı Notasyonlar

Notasyonlar (Annotations) Java 5’den beri Java ortamında kullanılan bileşenlerdir. Notasyonlar genel olarak bir bileşene özellik katma veya konfigürasyon amaçlı olarak kullanılmaktadır. Bu yazıda notasyonların genel özelliklerinden ve Java 8 *Repeated Annotations* yeniliğinden bahsedilecektir.

Notasyonlar `@` işaretiyle başlayan arayüzlerdir ve notasyonlar içinde alanlar tanımlanabilmektedir.

Örneğin;

@Single Notasyonu.

```
public @interface Single {

    String value();

}
```

Notasyonlar çeşitli alanlara uygulanabilmektedir. Bu alanlar;

Nereye uygulanabilir?	Açıklama
TYPE	Sınıf, arayüz, soyut sınıf başlarına
METHOD	Metod başlarına
FIELD	Global alan başlarına

Nereye uygulanabilir?	**Açıklama**
PARAMETER	Metod parametrelerine
CONSTRUCTOR	Constructor başına
ANNOTATION_TYPE	Notasyonların başına
PACKAGE	Paket deklarasyonu başına

Hangi notasyonun nereye veya nerelere uygulanabileceği, notasyonu yazan tarafından belirtilmektedir. Bu belirtim işlemi ise `@Target` isimli notasyon ile sağlanmaktadır.

Örnek 1.

```
@Target({ElementType.METHOD})
public @interface Single {

    String value();

}
```

`@Single` notasyonunun uygulanabilirlik alanı `METHOD` olarak tanımlandığı için, `@Single` notasyonu sadece metod başlarında kullanılabilir. Diğer alanlarda kullanılamaz, kullanılmaya kalkışılırsa derleme zamanında hata alınır.

Uygulama 1.

```
@Single // Buraya konamaz
public class App {

    @Single // Buraya konabilir
    public void doIt() {
```

```
    }
}
```

Örnek 2.

```
@Target({ElementType.METHOD,ElementType.TYPE})
public @interface Single {

    String value();

}
```

Şimdi ise `@Single` notasyonu hem metod başlarına hem de sınıf, arayüz, soyut sınıf gibi bileşenlerin başlarına eklenebilir.

Uygulama 2.

```
@Single // Artık buraya da konabilir
public class App {

    @Single // Buraya uygulanamaz :(
    String message = "Merhaba Dünya";

    @Single // Buraya konabilir
    public void doIt() {

    }
}
```

`@Target` notasyonu ile yapılan aslında, notasyonun uygulanabilirlik alanını, yani yetki alanını belirlemektir. Örneğin Uygulama 2'de uygulanabilirlik alanı olarak `METHOD` ve `TYPE` belirlenmiştir. Bu sebeple sadece bununla ilgili kısımlara `@Single` notasyonu uygulanabilir.Mesela örnekte olduğu gibi global alanlara uygulanamaz. Eğer buraya da uygulanabilir olması istenirse `FIELD` enum bileşeni `@Target` notasyonuna eklenmelidir.

Notasyonlar ve Alanları

Notasyonlar konfigürasyon amacıyla kullanılırken, barındırdığı değişken alanlarından faydalanılmaktadır. Bu alanların tipi, ismi ve gerekirse varsayılan değeri tanımlanabilmektedir.

Fakat bu alanlarda tip sınırlaması vardır, bu tipler şunlar olabilir;

String
Temel tipler
Class
Bir Enum tipi
Diğer bir notasyon tipi
Yukarıdaki tiplerin dizi tipleri

`@Single` notasyonunu düşünürsek `String` türünde `value` adında tek bir alana sahiptir. Değer ataması ise deklerasyon anında `()` içerisinde yapılmaktadır.

Örneğin;.

```
@Single(value="Merhaba Dünya")
public interface Hello {

}
```

Notasyonlar için `value` alanının özel bir anlamı vardır. Eğer başka bir alana değer ataması yapılmayacaksa `value` yazılmasa bile bu değer ataması `value` alanına yapılmaktadır.

Örneğin;.

```
@Single("Merhaba Dünya") ❶
public interface Hello {
```

```
}
```

❶ Denktir `@Single(value="Merhaba Dünya")`

Notasyonlara Erişim

Notasyonların nereye uygulandıkları ve içerisinde barındırdıkları veriler derleme zamanında (compile time) veya çalışma zamanında (runtime) elde edilebikmektedir. Fakat bir notasyona hangi zamanlarda erişilebileceği `@Retention` isimli notasyon ile belirtilmelidir. Retention (tutma) politikasının 3 tipi vardır.

SOURCE
: Notasyonlar derleme zamanında yok sayılır.

CLASS
: Notasyonlar derleme zamanında sınıf içerisinde bulundurulur, fakat çalışma zamanında bulunması zorunlu değildir. Varsayılan hal budur.

RUNTIME
: Notasyonlar çalışma zamanında erişilmek üzere sınıf içerisinde bulundurulur. Çalışma zamanında erişim Java Reflection API ile yapılır. Sık kullanılan hal budur.

Not
: Rastgele 10 notasyon belirleyip, @Retention notasyonunu incelediğinizde %90 üzeri ağırlıkta tutma politikasının `RUNTIME` olarak yapılandırıldığını görebilirsiniz.

Örneğin;.

```
@Target({ElementType.METHOD, ElementType.TYPE}) ❶
@Retention(RetentionPolicy.RUNTIME) ❷
public @interface Single {

    String value();

}
```

❶ Metod ve Sınıfların başına uygulanabilir.

❷ Çalışma zamanında bu notasyona erişilebilir.

Yukarıda `@Target` ve `@Retention` notasyonlarıyla yapılandırılmış kullanıma hazır bir `@Single` notasyonunu görüyoruz.

Notasyonların Tekrarlı Kullanılması

Java 8 öncesinde bir notasyon, iki kere aynı yerde kullanılamamaktaydı.

Hatalı kullanım örneği;.

```
@Single("Merhaba")
@Single("Jupiter")
public class App {


}
```

Örneğin `@Single` notasyonu yukarıdaki gibi aynı alanda kullanılamaz. Bu sınırlılık Java 8 öncesinde ikincil bir notasyon üzerinden aşılmaktaydı.

Örneğin;.

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Multiple {

    Single[] value();
}
```

Multiple notasyonu `Single` tipli diziler barındırabilen bir `value` alanına sahiptir. Artık Multiple notasyonu üzerinden birden fazla @Single notasyonu uygulanabilirdir.

Doğru kullanım örneği;.

```
@Multiple(value = {
    @Single("Merhaba"),
    @Single("Jupiter")
})
public class App {


}
```

NOTE : Notasyonlarda dizi tipli alanlara atamamalar, `{ }` arasında virgül , ile ayrılmış olarak yapılmaktadır.

Java 8 ile birlikte bu sınırlılık ortadan kalkmıştır. Bu sınırlılığı ortadan kaldırmak için `@Repeatable` notasyonundan faydalanılmaktadır.

```
@Repeatable(Multiple.class) // Dikkat

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Single {
    String value();
}
```

`@Repeatable` notasyonu tekrarlayacak notasyona uygulanmaktadır. `@Repeatable` ntoasyonunun value alanına ise, sarmalayıcı notasyonun sınıf tipi tanımlanmalıdır. Bu yolla tekrarlı notasyonlar alt alta, yan yan istendiği kadar tanımlanabilmektedir.

Örneğin;.

```
@Single("Merhaba")
@Single("Uranüs")
public class App {

  public static void main(String[] args) {

        Class<App> app = App.class; ❶
        Single[] notz = app.getAnnotationsByType(Single.class);
❷

        for (Single not : notz) { ❸
            System.out.println(not.value()); ❹
            // Merhaba
            // Uranüs
        }

    }

}
```

❶ App sınıfının sınıf tipini alır.

❷ App sınıfına uygulanmış tüm Single tipli notasyonları bulur.

❸ Tüm uygulanmış Single notasyonlarını dolaşır.

❹ O anki Single notasyonunun value() alanını çıktılar.

Tekrar görüşmek dileğiyle.

Bölüm 7. Method Reference

Java programlama dilinde metodlar, sınıfların içinde bulunan iş birimleridir.

Metodlara erişimin ise statik olup olmadığına göre iki erişim biçimi vardır. Statik metodlara sınıf üzerinden erişilebilirken, statik olmayan metodlara nesne üzerinden erişim sağlanabilmektedir.

Java 8 ile beraber metod birimleri, bir lambda ifadesine, dolayısıyla bir fonksiyonel arayüze karşılık olarak referans verilebilmektedir.

Lambda ifadeleri yazılırken, tek metoda sahip arayüzün (fonksiyonel arayüz) metod girdi ve çıktısı baz alınmaktadır. Eğer daha önce yazdığınız bir metodun girdi ve çıktısı, bir fonksiyonel arayüz metodunun girdi ve çıktısına birebir uyuyorsa, o metod bir lambda deyimi yerine kullanılabilmektedir.

Örneğin;

Consumer arayüzü `T` türünde tek bir parametre alır. Metod dönüşü `void` 'dir.

```
@FunctionalInterface
public interface Consumer<T> {

    void accept(T t); // t-> {}

}
```

Yukarıdaki `Consumer` arayüzü türünden bir nesneyi, `e → {}` lambda deyimiyle oluşturabiliriz.

```
Consumer<String> consumerWithLambda = e-> {

};
```

Lambda ifadeleri yazarken metod girdi ve çıktısına odaklandığımıza göre, lambda deyimiyle birebir örtüşen metodları, lambda deyimleri yerine kullanabilir miyiz? Bunu aşağıdaki metod üzerinden düşünelim.

```
public void herhangiBirMetod(String e){

}
```

Yani `herhangiBirMetod` metodunun imzasıyla `Consumer<String>` arayüzünün accept metodunun imzası birbiriyle aynıdır. `herhangiBirMetod`, String türünden tek bir parametre almaktadır, metod dönüşü ise `void` 'dir. Bu uyum neticesinde eğer istenirse, bu metodun referansı ile `Consumer<String>` türünden bir nesne oluşturulabilir.

Bir metodu referans vermek

Java programlama dilinde bir metodun nasıl referans verileceği metodun statik olup olmadığına göre değişmektedir. Bir metodun referans olarak verilebilmesi için ise `::` ikilisi kullanılır.

Statik metodlarda. Statik metodlar sınıfa ait bileşenlerdir. Bu yüzden erişim kurulurken dahili olduğu sınıf üzerinden erişim kurulur.

Örneğin;.

```
Consumer<String> consumerWithMetRef =
<ClassName>::herhangiBirMetod;
```

Statik olmayan metodlarda. Statik olmayan metodlar nesneye ait bileşenlerdir. Bu sebeple nesnenin referansı üzerinden erişilmelidirler.

Örneğin;.

```
Consumer<String> consumerWithMetRef =
<ObjectRef>::herhangiBirMetod;
```

Örnek 1. Şimdi birkaç örnek ile metod referansları deneyimleyelim. Aşağıda `"Ali"`,`"Veli"`,`"Selami"` verilerini içeren bir List nesnesi bulunmaktadır. Bu nesnenin ise, `forEach` isimli metodu bulunmaktadır. Bu nesnenin kabul ettiği parametre ise `Consumer<String>` türündendir.

```
List<String> names= Arrays.asList("Ali,Veli,Selami");

names.forEach(new Consumer<String>() { ❶
```

```
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
});
```

❶ Sırayla konsole çıktılar.

Şimdi bu örneği `Lambda` ifadesiyle yeniden aşağıdaki gibi yazabiliriz.

```
List<String> names= Arrays.asList("Ali,Veli,Selami");

names.forEach(s->{
   System.out.println(s);
});
```

Lambda ifadesi, satır sayısı bazında kodu iyice kısalttı. Şimdi bu aynı işi metod referans kullanarak yapalım.

```
public class App{

    static List<String> names=
Arrays.asList("Ali,Veli,Selami");

    // Statik metod
    public static void herhangiBirMetod(String s){
        System.out.println(s);
    }

    // Non-statik metod
    public void digerBirMetod(String s){
        System.out.println(s);
    }


    public static void main(String[] args){

        names.forEach(App::herhangiBirMetod); ❶

        // veya

        App app=new App();
        names.forEach(app::digerBirMetod); ❷
```

```
        // veya

        names.forEach(System.out::println); ❸
    }

}
```

❶ Statik metodu referans vermek

❷ Non-statik metodu referans vermek

❸ Diğer bir örnek

Örnek 2. `Collections#sort` metodu ile bir dizi nesneyi sıralamak istiyoruz diyelim. Bu sıralamayı, metodun 2. parametresinde `Comparator` türünden bir nesne ile kontrol edebiliriz.

```
List<Integer> numbers = Arrays.asList(5, 10, 1, 5, 6);

Collections.sort(numbers, new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o1 - o2;
    }
});
```

Not

Fonksiyonel olarak `Comparator` arayüzü ve `java.util.BiFunction` arayüzü birebir aynıdır.

Şimdi anonim olarak oluşturulan `Comparator` türünden nesneyi Lambda ifadeleriyle yeniden yazalım.

```
List<Integer> numbers = Arrays.asList(5, 10, 1, 5, 6);
```

```
Collections.sort(numbers, (o1, o2) -> o1 - o2)
```

Şimdi ise Lambda deyimi yerine metod referans kullanarak bu işi yapalım.

```
public Integer sirala(Integer o1, Integer o2){

    return o1 - o2;

}

public void birMetod(){

    List<Integer> numbers = Arrays.asList(5, 10, 1, 5, 6);

    Collections.sort(numbers, this::sirala); ❶
}
```

❶ Lambda yerine metod referansı

Metod referans kullanmak var olan iş mantığını kolay bir biçimde referans vermeye olanak sağlamaktadır.

Tekrar görüşmek dileğiyle.

Bölüm 8. Default Methods

Java 8 ile beraber varsayılan metod özelliği bir dil özelliği olarak Java'ya katıldı. Varsayılan metodların literatürde birçok farklı isim ile anılmaktadır. Bunlar;

- Default method
- Defender method
- Virtural extension method

Java 8 evvelinde arayüz bileşenlerinde dilin tasarımı açısından sadece soyut metodlar bulunabilmekteydi. Somut yani gövdeli metodlar bulunamamaktaydı. Aşağıda doğru ve yanlış kullanımlara birer örnek görmekteyiz.

Doğru bir kullanım örneği.

```
public interface Arac {
    void gazla(); ❶
}
```

❶ Soyut, gövdesiz metod, doğru kullanım.

Yanlış bir kullanım örneği.

```
public interface Arac {
    void gazla() {
     // bla bla bla
    }; ❶
}
```

❶ Somut, gövdeli metod, yanlış kullanım.

Varsayılan Metoda Giriş

Java 8 ile birlikte bir arayüz bileşeninde bir yada daha fazla sayıda defender method tanımlanabilmektedir. Varsayılan metodlar `default` anahtar kelimesiyle tanımlanmaktadır. Örneğin;

```
public interface Arac {

    default void gazla(){ // Defender method ❶

        System.out.println("Araç:  çalışıyor..");

    }

    void dur(); // Soyut metod ❷
}
```

❶ numaralı kısımdaki `gazla()` metodu `default` anahtar ifadesi aracılığıyla bir varsayılan metoda dönüştürülmüştür. Varsayılan metodlar arayüzlere iş mantığı yerleştirmeye müsade eden özel metodlardır.

❷ numaralı kısımda ise olağan biçimde gövdesiz bir metod bulunmaktadır.

Varsayılan metoda sahip bir arayüzden türeyen alt sınıflar, arayüzün sahip olduğu tüm defender metodları tüketebilmektedir.

Arac türünden Minibus sınıfı.

```
public class Minibus implements Arac {

    @Override
    public void dur() {
        System.out.println("Minibüs duruyor..");
    }


}
```

Örneğin yukarıda yer alan `Minibus` sınıfı `Arac`arayüzü türünden bir sınıftır. Bu sebeple, Minibus sınıfı türünden nesneler `Arac` arayüzü içerisindeki `gazla()` metodunu koşturabilecektir.

```
Minibus minibus=new Minibus();
minibus.gazla(); ❶
minibus.dur(); ❷
```

❶ Arac içindeki `gazla()` varsayılan metodu koşturuluyor.

❷ Minibus içindeki `dur()` metodu koşturuluyor.

```
Araç:  çalışıyor..
Minibüs duruyor..
```

Yukarıda görüldüğü üzere, normalde `Minibus` sınıfı içerisinde `gazla()` metodu bulunmamasına rağmen, `Arac` arayüzü içindeki defender metodu koşturabildi.

Varsayılan metodlarda çakışma

Eğer bir Java sınıfı, aynı isimde varsayılan metoda sahip birden fazla arayüzü uygularsa, derleme zamanında hata ile karşılaşılır.

Aynı isimde varsayılan metodlara sahip iki arayüz.

```
public interface Arac { ❶

    default void gazla(){
        System.out.println("Araç:  çalışıyor..");
    }

    void dur();
}

public interface Tasit { ❷

    default void gazla(){
        System.out.println("Taşıt: çalışıyor..");
    }
}
```

Örneğin **(1)** numarada `Arac`, **(2)** numarada da `Tasit` arayüzleri `gazla()` adında varsayılan metodlara sahiptir.

Şimdi bu iki türü birden uygulayan bir `Otobus` sınıfı yazalım.

Çakışma durumu örneği.

```
public class Otobus implements Arac, Tasit {

    @Override
    public void dur() {
        System.out.println("Araç duruyor..");
    }
}
```

Otobus sınıfı bu haliyle derlendiğinde aşağıdaki derleme hatası alınacaktır.

Çakışma durumu hata mesajı.

```
Error:(6, 8) java: class com.kodcu.def.Otobus inherits
unrelated defaults for gazla() from types com.kodcu.def.Arac
and com.kodcu.def.Tasit
```

Çünkü ortada Otobus sınıfının hangi gazla() metodunu koşturacağına dair bir ikilem vardır. JVM ikilem durumlarını hiç sevmez, ona bir seçim şansız sunmalıyız. Bu çakışma durumu, varsayılan metodu Otobus sınıfı içinde yeniden düzenlenerek (@Override ederek) giderilebilmektedir.

Çakışma durumunun giderilmesi - 1.

```
public class Otobus implements Arac, Tasit {

    @Override
    public void dur() {
        System.out.println("Araç duruyor..");
    }

    @Override
    public void gazla() {
        System.out.println("Otobüs çalışıyor..");
    }
}
```

Otobus sınıfına gazla() metodu ekleyerek yeniden düzenlendiğinde artık çakışma durumu giderilmiş durumdadır. Sınıf bu haliyle Otobüs çalışıyor.. mesajını çıktılayacaktır.

Fakat burada farklı bir ihtiyaç daha göze batmaktadır. Bu durumda Arac veya Tasit arayüzleri içindeki çakışan gazla() metodları nasıl alt sınıflarda kullanılabilir?

İşte bu noktada <arayüz-adı>.super.<metod-adı>() biçimi ile arayüzlerdeki varsayılan metodlar çakışma olmadan koşturulabilmektedir.

Çakışma durumunun giderilmesi - 2.

```
public class Otobus implements Arac, Tasit {

    @Override
    public void dur() {
        System.out.println("Araç duruyor..");
    }
```

```
    @Override
    public void gazla() {

        Arac.super.gazla(); ❶
            /* veya */
        Tasit.super.gazla(); ❷
    }
}
```

❶ Arac arayüzünün `gazla()` metodunu koşturur

❷ Tasit arayüzünün `gazla()` metodunu koşturur

Varsayılan metodlar ve Fonksiyonel arayüzler

Fonksiyonel arayüzler, tek bir gövdesiz metoda sahip özel arayüzlerdir. Eğer bir Java arayüzünün içinde birden fazla sayıda soyut metod varsa, bu arayüzler fonksiyonel arayüz olamamaktadır. Fonksiyonel arayüzlerin en önemli özelliği, Lambda ifadesi olarak temsil edilebilmesidir.

Arayüzler içinde tanımlanan varsayılan metodlar ise, bir arayüzün fonksiyonel olabilmesini etkilememektedir. Çünkü yazılan Lambda deyimleri, arayüz içindeki tek bir soyut metoda odaklı olarak dönüştürülmektedir.

```
public interface Arac {

    default void gazla(){
        System.out.println("Araç:  çalışıyor..");
    }

    void dur();
}
```

Örneğin yukarıdaki `Arac` arayüzünü fonksiyonel arayüz olabilirliği açısından değerlendirelim. Arac sınıfının fonksiyonel arayüz olabilmesi için tek bir soyut metoda sahip olması gerekmektedir. Arac arayüzü içindeki `dur()` metodu soyuti gövdesiz bir metod olduğu için bir fonksiyonel arayüzdür. `gazla()` metodu ise bir varsayılan metod olduğundan fonksiyonel olabilirliğe bir etkisi yoktur. Bu noktada, Arac arayüzü bir fonksiyonel arayüz olduğundan Lambda deyimi olarak yazılabilecektir. Tabi ki Lambda deyimi, `dur()` isimli soyut metod dikkate alınarak yazılmalıdır.

```
Arac otobus = ()-> System.out.println("Otobüs duruyor.."); ❶
otobus.gazla();
otobus.dur();
```

```
Araç:  çalışıyor..
Otobüs duruyor..
```

Yukarıda **(1)** numarada yazılı Lambda deyimi, dur() metoduna karşılık olarak tanımlanmıştır. Bu sebeple Arac arayüzü türünden bir nesne

oluşturmaktadır.

Varsayılan metodlar ve JDK

JDK 1.8 içerisinde bazı noktalarda varsayılan metodlar kullanılmaktadır. `java.util.Collection` arayüzünde bunu fazlaca görmekteyiz.

```
public interface Collection<E> extends Iterable<E> {

    ...

    default boolean removeIf(Predicate<? super E> filter) {
        Objects.requireNonNull(filter);
        boolean removed = false;
        final Iterator<E> each = iterator();
        while (each.hasNext()) {
            if (filter.test(each.next())) {
                each.remove();
                removed = true;
            }
        }
        return removed;
    }

    ...

    @Override
    default Spliterator<E> spliterator() {
        return Spliterators.spliterator(this, 0);
    }

    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }

    default Stream<E> parallelStream() {
        return StreamSupport.stream(spliterator(), true);
    }
}
```

`Collection` sınıfı içindeki `removeIf`, `stream`, `parallelStream` ve `spliterator` metodları varsayılan metodlardır. Bu sebeple `Collection` türünden tüm nesneler bu varsayılan metodları miras alarak tüketebilmektedir.

`Collection` arayüzünde olduğu gibi `Iterable` arayüzünde de varsayılan metod bulunduğunu görebiliyoruz.

```
@FunctionalInterface
public interface Iterable<T> {

    Iterator<T> iterator();

    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
                action.accept(t);
        }
    }
}
```

`Iterable#forEach` varsayılan metodu sayesinde Iterable türünden tüm veri tipleri, bu metodu ortak olarak tüketebiliyor.

Tekrar görüşmek dileğiyle..

Bölüm 9. Stream API

Java 8 içerisinde yığınsal verileri kolay işlemek açısından Stream API yeniliği getirilmiştir.

Stream API yığınsal veriler üzerinde çalışmaktadır. Yığınsal veri deyince ilk akla gelen hiç şüphesiz **diziler** (`byte[]`,`String[]` gibi) ve Java **Collection API** bileşenleridir (`List`,`Set` gibi)

Stream API, bu gibi yığınsal veriler üzerinde çeşitli sık kullanılan operasyonları kolay, özlü ve verimli bir biçimde koşturmaya olanak tanımaktadır. Bu operasyonlardan en sık kullanılanları aşağıdaki gibidir.

Metod	**Açıklama**
filter	Filtreleme
forEach	iterasyon
map	Dönüştürme
reduce	İndirgeme
distinct	Tekilleştirme
sorted	Sıralama
limit	Aralık alma

Metod	Açıklama
collect	Türe dönüşüm
count	Sayma
…	

Bu operasyonlar ve daha fazlası java.util.stream.Stream arayüzü içinde bulunmaktadır. Stream arayüzünün sadeleştirilmiş hali aşağıdaki gibidir.

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {

    Stream<T> filter(Predicate<? super T> predicate);
    <R> Stream<R> map(Function<? super T, ? extends R> mapper);
    IntStream mapToInt(ToIntFunction<? super T> mapper);
    LongStream mapToLong(ToLongFunction<? super T> mapper);
    DoubleStream mapToDouble(ToDoubleFunction<? super T>
mapper);
    <R> Stream<R> flatMap(Function<? super T, ? extends
Stream<? extends R>> mapper);
    IntStream flatMapToInt(Function<? super T, ? extends
IntStream> mapper);
    LongStream flatMapToLong(Function<? super T, ? extends
LongStream> mapper);
    DoubleStream flatMapToDouble(Function<? super T, ? extends
DoubleStream> mapper);
    Stream<T> distinct();
    Stream<T> sorted();
    Stream<T> sorted(Comparator<? super T> comparator);
    Stream<T> peek(Consumer<? super T> action);
    Stream<T> limit(long maxSize);
    Stream<T> skip(long n);
    void forEach(Consumer<? super T> action);
    void forEachOrdered(Consumer<? super T> action);
    Object[] toArray();
    <A> A[] toArray(IntFunction<A[]> generator);
    T reduce(T identity, BinaryOperator<T> accumulator);
    Optional<T> reduce(BinaryOperator<T> accumulator);
    <U> U reduce(U identity,
                 BiFunction<U, ? super T, U> accumulator,
```

```
                 BinaryOperator<U> combiner);
    <R> R collect(Supplier<R> supplier,
                  BiConsumer<R, ? super T> accumulator,
                  BiConsumer<R, R> combiner);
    <R, A> R collect(Collector<? super T, A, R> collector);
    Optional<T> min(Comparator<? super T> comparator);
    Optional<T> max(Comparator<? super T> comparator);
    long count();
    boolean anyMatch(Predicate<? super T> predicate);
    boolean allMatch(Predicate<? super T> predicate);
    boolean noneMatch(Predicate<? super T> predicate);
    Optional<T> findFirst();
    Optional<T> findAny();
}
```

Stream nesnesi nasıl elde edilir?

Stream türünden nesneler çeşitli yollarla elde edilebilmektedir.

Collection API ile

Collection arayüzü türünden türeyen tüm nesneler, stream() veya parallelStream() metodlarını çağırarak Stream<E> türünden bir nesne elde edebilmektedir.

```
public interface Collection<E> extends Iterable<E> {

...

    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }

    default Stream<E> parallelStream() {
        return StreamSupport.stream(spliterator(), true);
    }

...

}
```

stream() metodu ile elde edilen Stream nesnesi yapacağı işlemleri ardışıl olarak yaparken, parallelStream() metoduyla elde edilen Stream nesnesi,

bazı operasyonları paralel olarak koşturabilmektedir.

Örneğin;.

```
List<String> names = Arrays.asList("Ali","Veli","Selami"); ❶

Stream<String> stream = names.stream(); ❷
Stream<String> parallelStream = names.parallelStream(); ❸
```

❶ Collection türünden bir nesne

❸ Ardışık stream

Paralel stream

New I/O ile

Java içerisindeki bazı I/O sınıfları üzerinden Stream nesneleri elde edilebilmektedir.

```
Path dir = Paths.get("/var/log"); ❶

Stream<Path> pathStream = Files.list(dir); ❷
```

❶ `/var/log` dizinine denk gelen bir Path nesnesi

❷ Files#list metodu üzerinden bir Stream<Path> nesnesi

IntStream, DoubleStream, LongStream ile

Stream arayüzü BaseStream arayüzünden türemektedir. Stream arayüzüne benzer biçimde IntStream, DoubleStream ve LongStream arayüzleri de

BaseStream arayüzünden türemektedir.

Stream arayüzü türünden nesneler tüm veri tipleriyle çalışmak için oluşturulan bir arayüzken, buradaki üç eleman ise, sadece sınıf başındaki tip ile özel olarak çalışmak için oluşturulan arayüzlerdir.

Örneğin;.

```
IntStream intOf = IntStream.of(1, 2, 3); ❶
IntStream intRange = IntStream.range(1, 10); ❷

DoubleStream doubleOf = DoubleStream.of(1.0, 3.5, 6.6); ❸

LongStream longOf = LongStream.of(3, 5, Long.MAX_VALUE,9); ❹
LongStream longRange = LongStream.range(1, 100); ❺
```

❶ (1,2,3) içeren IntStream nesnesi

❷ (1,…,10) arasını içeren IntStream nesnesi

❸ (1.0, 3.5, 6.6) içeren DoubleStream nesnesi

❹ (3, 5, Long.MAX_VALUE,9) içeren LongStream nesnesi

❺ (1,…,100) arasını içeren LongStream nesnesi

Stream API Örnekleri

Bu kısımda çeşitli Stream API metodları ile küçük uygulamalar yer almaktadır.

forEach

Stream içerisindeki yığınsal veriyi tek tek tüketmek için yapılandırılmıştır. Consumer arayüzü türünden bir parametre bekler.

```
List<String> names =
Arrays.asList("Ali","Veli","Selami","Cem","Zeynel","Can","Hüsey
in");

Stream<String> stream = names.stream();

stream.forEach(name -> {
    System.out.println(name);
});

// veya stream.forEach(System.out::println);
```

filter

Stream içerisindeki yığınsal veri üzerinde süzme işlemi yapar. Predicate arayüzü türünden bir parametre ile filtreleme işlemini yapar.

```
List<String> names = Arrays.asList("Ali", "Veli", "Selami",
"Cem", "Zeynel", "Can", "Hüseyin");

Stream<String> stream = names.stream(); ❶

Predicate<String> predicate = name -> name.length() < 4; ❷

Stream<String> filtered = stream.filter(predicate); ❸

filtered.forEach(System.out::println); ❹
```

❶ Stream nesnesi elde ediliyor.

❷ Predicate sorgusu hazırlanıyor

❸ Süzme işlemi yapılıyor, yeni bir Stream nesnesi sunuluyor.

❹ Listeleniyor. [Ali, Cem, Can]

Not

Stream nesneleri tek kullanımlıktır. Stream nesnesinin çoğu metodu yeni bir Stream nesnesi sunmaktadır. Bu sebeple tüm operasyonlar zincirlemeli olarak yapılabilmektedir.

Örneğin;.

```
names
    .stream()
    .filter(name -> name.length() == 4)
    .forEach(System.out::println);
```

distinct

Bir Stream içerisinden tekrarlı veriler çıkarılmak isteniyorsa distinct metodundan faydalanılabilir.

```
IntStream stream = IntStream.of(1, 1, 2, 3, 5, 8, 13, 13, 8); ❶

stream
    .distinct()
    .forEach(System.out::println); ❷
```

❶ IntStream nesnesi

❷ [1,2,3,5,8,13]

sorted

Stream içerisindeki yığınsal verinin sıralanmış Stream nesnesini döndürür.

```
IntStream stream = IntStream.of(13, 1, 3, 5, 8, 1, 13, 2, 8); ❶

stream
        .sorted()
        .forEach(System.out::println); ❷
```

❶ IntStream nesnesi

❷ [1,1,2,3,5,8,8,13,13]

limit

Bir Stream yığını içerisindeki ilk N veri barındıran yeni bir Stream nesnesi sunmaktadır.

```
LongStream range = LongStream.range(1, 10000); ❶

        range
                .limit(10)
                .forEach(System.out::println); ❷
```

❶ (1,…,10000) arasını içeren bir Stream

❷ İlk 10 veri : [1,…,10]

count

Stream içerisindeki eleman sayısını hesaplar.

```
IntStream range = IntStream.range(1, 10);
IntStream rangeClosed = IntStream.rangeClosed(1, 10);

System.out.println(range.count()); ❶
System.out.println(rangeClosed.count()); ❷
```

❶ 9

❷ 10

collect

Stream türünden nesneler, yığın verileri temsil eden özel nesnelerdir. Fakat Stream biçimi bir veri yapısı sunmamaktadır. collect metodu ağırlıklı olarak , Stream nesnelerini başka biçimdeki bir nesneye, veri yapısına dönüştürmek için kullanılmaktadır.

`Stream#collect` metodu Collector türünden bir parametre kabul etmektedir. Bu parametre ile istendik türe dönüşüm sağlanmaktadır. Collector türünden arayüzler, `Collectors` sınıfının çeşitli statik metodlarıyla elde edilebilmektedir.

```
List<String> names = Arrays.asList("Ali", "Veli", "Selami",
"Veli", "Selami", "Can", "Hüseyin");

List<String> list =
names.stream().collect(Collectors.toList()); ❶

Set<String> set = names.stream().collect(Collectors.toSet()); ❷

Long count = names.stream().collect(Collectors.counting()); ❸

String collect = names.stream().collect(Collectors.joining(" -
")); ❹

Map<Integer, List<String>> integerListMap =
names.stream().collect(Collectors.groupingBy(name ->
name.length())); ❺
```

❶ Stream nesnesinden List nesnesi üretir. List["Ali", "Veli", "Selami", "Veli", "Selami", "Can", "Hüseyin"]

❷ Stream nesnesinden Set nesnesi üretir. Set["Ali", "Veli", "Selami","Can", "Hüseyin"]

❸ Stream nesnesinin eleman sayısını üretir. **7**

❹ Stream nesnesini birleştirir. Ali - Veli - Selami - Veli - Selami - Can - Hüseyin

❺ Stream nesnesini isim uzunluğuna göre gruplar.

Tablo 9.1. Map<Integer, List<String>> nesnesinin temsili tablo görünümü

Key	**Value**
3	Ali
	Can
4	Veli
	Veli
6	Selami
	Selami

Key	Value
7	Hüseyin

map

Stream içindeki yığınsal olarak bulunan her bir veriyi dönüştürmeye olanak tanır. Dönüştürüm işlemi Stream içerisindeki her bir öğe için ayrı ayrı yapılmaktadır. Stream#map metodu Function türünden bir parametre beklemektedir.

Örnek 1;. Bir `List<String>` içindeki her bir öğenin harflerini büyütelim.

```
List<String> names = Arrays.asList("Ali", "Veli", "Selami",
"Cem");

Stream<String> stream = names.stream(); ❶
Stream<String> upperStream= stream.map(name ->
name.toUpperCase()); ❷
List<String> upperNames =
upperStream.collect(Collectors.toList()); ❸
```

❶ Stream<String> nesnesi elde ediliyor

❷ Her bir ismin harfleri büyütülüyor

❸ List["ALİ","VELİ","SELAMİ","CEM"]

Örnek 2;. 1,5 arası sayıların karelerini hesaplayalım.

```
IntStream
        .rangeClosed(1, 5)
        .map(n -> n*n)
        .forEach(System.out::println); ❶
```

❶ [1, 4, 9, 16, 25]

reduce

Bir Stream içerisindeki verilerin teker teker işlenmesidir. Teker teker işleme sürecinde, bir önceki adımda elde edilen sonuç bir sonraki adıma girdi olarak sunulmaktadır. Bu sayede yığılmlı bir hesaplama süreci elde edilmiş olmaktadır.

Stream#reduce metodu ilk parametrede `identity` değeri, ikinci parametrede ise BinaryOperator türünden bir nesne kabul etmektedir.

reduce işleminde bir önceki hesaplanmış değer ile sıradaki değer bir işleme tabi tutulmaktadır. İşleme başlarken bir önceki değer olmadığı için bu değer `identity` parametresinde tanımlanmaktadır.

Örnek 1;. 1,2,3,4,5 sayılarının toplamını hesaplayalım.

```
int result = IntStream
                .of(1, 2, 3, 4, 5)
                .reduce(0, (once, sonra) -> {
                    System.out.format("%d - %d %n", once,
sonra);
                    return once + sonra;
                });
```

Toplama işleminde 0 etkisiz eleman olduğu için, identity değeri 0 seçildi.

Uygulama çalıştırıldığında 15 sonucu elde edilir. reduce içindeki Lambda ifadesinde ise aşağıdaki çıktı elde edilir.

```
 0 - 1
 1 - 2
 3 - 3
 6 - 4
10 - 5
```

Önce hesaplanmış değeri, **Sonra** ise sıradaki değeri temsil etmektedir. Bir adımda çıkan hesaplamanın sonucu, bir sonraki adımda (satırda) **Önce** sütununa sunulmaktadır.

Önce	Sonra	Hesaplama
0	1	0+1 ↵
1	2	1+2 ↵
3	3	3+3 ↵
6	4	6+4 ↵
10	5	10+5 = **15**

Örnek 2;. 1,2,3,4,5 sayılarının çarpımını hesaplayalım.

```
// Lambda ile
int result = IntStream
                .of(1, 2, 3, 4, 5)
                .reduce(1, (once, sonra) -> once*sonra);

// veya Method reference ile
result = IntStream
            .of(1, 2, 3, 4, 5)
            .reduce(1, Math::multiplyExact);
```

map & reduce

map ve reduce işlemleri birlikte kullanımı çok fazla tercih edilen iki operasyondur. Bu operasyonları önemli kılan ise, bu iki operasyonun dağıtık sistemler için çok uygun olmasıdır. Piyasada Map & Reduce işlemlerini dağıtık mimarilerde kullanan birçok teknoloji bulunmaktadır. Tabiki Java 8 ile kullandığımız map & reduce ikilisi tek JVM üzerinde koştuğu için dağıtık değildir.

Örneğin;

- Hazelcast
- Hadoop
- MongoDB gibi.

Örnek 1;. Elimizde Person sınıfı türünden 5 nesne bulunsun. Bu 5 nesne içinden tüm kişilerin yaşlarının ortalamasını hesaplamak isteyelim. Böyle bir senaryo için map & reduce metodlarını birlikte tercih edebiliriz.

```
public class Person {
    private String name;
    private Integer age;

   // getter, setter ve constructor metodları
}

Person p1 = new Person("Ahmet", 12);
Person p2 = new Person("Ali", 20);
Person p3 = new Person("Ayşe", 30);
Person p4 = new Person("Murat", 51);
Person p5 = new Person("Zeynep", 60);

List<Person> personList = Arrays.asList(p1, p2, p3, p4, p5); ❶

personList
        .stream() ❷
        .map(p -> p.getAge()) ❸
        .map(Double::valueOf) ❹
        .reduce(0, (a, b) -> (a + b)/2); ❺
```

❶ Person listesi

❷ Stream nesnesi elde ediliyor

❸ Nesnenin yaş alanına göre mapping yapılıyor.

❹ Integer → Double dönüşümü

❺ Ortalamalar hesaplanıyor

Örnek 2;. Person listesinde bazı kişilerin yaş alanları null değer içersin. Bu durumda çalışma zamanında nullpointerexception istisnası elde edilecektir. Bu gibi bir durumda filtreleme yapısını işlemimize ekleyebiliriz.

```
Person p1 = new Person("Ahmet", 12);
Person p2 = new Person("Ali", null);
Person p3 = new Person("Ayşe", 30);
Person p4 = new Person("Murat", null);
Person p5 = new Person("Zeynep", 60);

List<Person> personList = Arrays.asList(p1, p2, p3, p4, p5);

personList
        .stream()
        .filter(Objects::nonNull) // Dikkat !!
        .map(p -> p.getAge())
        .map(Double::valueOf)
        .reduce(0, (a, b) -> (a + b)/2);
```

Parallel Stream

Stream arayüzü içindeki metodlardan ardışık işletilmesi gerekmeyenler, istenirse, CPU üzerinde paralel olarak koşturulabilmektedir. Bu sayede CPU çekirdeklerini tam verimli olarak kullanmak mümkün olmaktadır.

Stream API içerisinde paralel Stream elde etmek oldukça kolaydır.

Örneğin

```
List<Integer> ints = Arrays.asList(1, 3, 5, 7, 9, 11, 13, 15);

Stream<Integer> stream = ints.stream();
Stream<Integer> parallelStream = ints.parallelStream();
```

Collection#stream() metoduyla ardışıl (sequential) , Collection#parallelStream() metoduyla da paralel Stream nesnesi elde edilmektedir. Elde edilen paralel Stream nesnesiyle koşturulan işlemler paralel olarak koşabilmektedir.

Aynı zamanda bir ardışıl Stream nesnesinden paralel Stream nesnesi elde edilebilmektedir. Bunun için Stream#parallel metodu kullanılmaktadır.

Örneğin;.

```
List<Integer> ints = Arrays.asList(1, 3, 5, 7, 9, 11, 13, 15);

Stream<Integer> stream = ints.stream(); // Ardışıl
Stream<Integer> parallelStream = stream.parallel(); // Paralel
```

Aynı zamanda bir paralel Stream nesnesinden ardışıl Stream nesnesi de elde edilebilmektedir. Bunun için Stream#sequential metodu kullanılmaktadır.

Örneğin;.

```
List<Integer> ints = Arrays.asList(1, 3, 5, 7, 9, 11, 13, 15);

Stream<Integer> parallelStream = ints.parallelStream(); //
Paralel
Stream<Integer> stream = stream.sequential(); // Ardışıl
```

Örnek. Aşağıda bir dizi sayısal ifadeyi filtreleyen, sıralayan ve çıktılayan bir kod parçası görmekteyiz. Ayrıca bu işlemlerin paralel Stream nesnesiyle yapılmak istendiğini görüyoruz.

```
List<Integer> ints = Arrays.asList(1, 5, 3, 7, 11, 9, 15, 13);

 ints
    .parallelStream() // Paralel Stream
    .filter(Objects::nonNull) // null değilse
    .filter(n -> n > 0) // pozitif sayı ise
    .sorted() // sırala
    .forEach(System.out::println); // çıktıla
```

Bu örnekte filter ve sorted paralel olarak koşturulabilirdir. Fakat forEach metodu doğası gereği öğeleri ardışık çıktılamalıdır. İşte tam da bu adımda elimizdeki paralel Stream nesnesi ardışıl Stream nesnesine dönüştürülmektedir ve ardından forEach işlemini koşturmaktadır.

Yani elimizde paralel Stream nesnesi varsa, bu zincirlemeli işlemin her adımında paralel koşturma yapılacağı anlamını taşımamaktadır.

Lazy & Eager operasyonlar

Literatürde Lazy bir işlemin geç, ötelenmiş olarak yapılması iken, Eager ise yapılacak işlemin emir verilir verilmez yapılmasını temsilen kullanılır.

Stream API içerisindeki bazı operasyonlar Lazy bazıları ise Eager olarak koşturulmaktadır. Lazy davranışlı olan zincirli görevler, bir Eager operasyona gelene kadar koşturulmamaktadır.

```
List<Integer> names = Arrays.asList(1,2,3,6,7,8,9);

Stream<Integer> stream = names
    .stream()
    .filter(Objects::nonNull)
    .filter(n->n%2==1)
    .map(n->n*2);
```

Örneğin yukarıdaki liste üzerinde yapılmak istenen 2 `filter` ve 1 `map` işlemi Lazy işlemlerdir. Kod parçası bu haliyle çalıştırıldığında ne bir filtreleme ne de bir dönüştürme işlemi yapılacaktır. Burada yapılan sadece Stream nesnesini hazırlamaktır. Lazy işlemler gerekmedikçe işleme konulmamaktadır.

```
List<Integer> names = Arrays.asList(1,2,3,6,7,8,9);

Stream<Integer> stream = names
    .stream()
    .filter(Objects::nonNull) ❶
    .filter(n->n%2==1) ❷
    .map(n->n*2) ❸

stream.forEach(System.out::println); // Dikkat !! ❹
```

❶ Lazy

❷ Lazy

❸ Lazy

❹ Eager

Fakat bu hazırlanan Stream nesnesi, yukarıdaki gibi bir Eager operasyonla karşılaşırsa, önceki zincirlerde biriken Lazy işlemleri de harekete geçirecektir. Yani **(4)** numaradaki işlem, **(1)(2)(3)** numaralı işlemlerin tetikleyicisi konumundadır.

Tekrar görüşmek dileğiyle..

Bölüm 10. Java 8 ve JVM Dilleri

Java Sanal Makinesi (JVM), Java 7 ile başlayan Da Vinci Machine projesiyle, özellikle dinamik tipli dilleri JVM üzerinde çalışabilir kılmaktadır.

Sun Microsystem'in ilk adımlarını attığı bu proje, Oracle firmasıyla beraber de önem verilen bir konu olmaya devam etmektedir. JVM içerisinde statik tipli dilleri çalıştırabilmenin birden fazla amacı bulunmaktadır. Bunlar;

- JIT (Just in Time) Compiler ile yüksek performans sunmak
- Birçok dilin çalıştırılmasıyla JVM'i Polyglot bir ortam haline getirmek
- Farklı dil ve ekosistemleri Java ekosistemine yakınlaştırmak
- Farklı dil ekosistemlerinin gücünü JVM'de birleştirmek

JVM Dilleri

Halihazırda Java Sanal Makinesi üzerinde birçok programlama dili çalıştırılabilmektedir. Bu diller Tablo 10.1, "JVM Dilleri Tablosu"nda olduğu gibidir;

Tablo 10.1. JVM Dilleri Tablosu

Dil	Uygulayıcı kütüphane
Ada	JGNAT
BBx	BBj is a superset of BBx, PRO/5, and Visual PRO/5.
C	C to Java Virtual Machine compilers
CFML	Adobe ColdFusion

Dil	Uygulayıcı kütüphane
	Railo
	Open BlueDragon
Common Lisp	Armed Bear Common Lisp
	CLforJava
JavaScript	Rhino
	Nashorn
Pascal	Free Pascal
	MIDletPascal
Perl 6	Rakudo Perl 6
Prolog	JIProlog
	TuProlog
Python	Jython

Dil	Uygulayıcı kütüphane
REXX	NetRexx
Ruby	JRuby
Scheme	Bigloo
	Kawa
	SISC
	JScheme
Tcl	Jacl

Kaynak: List of JVM languages

Tablo 10.1, "JVM Dilleri Tablosu"nda listelenen programlama dilleri JVM bünyesinde koşturulabilmektedir. Bazı diller yorumlama usülüyle koşturulurken, bazıları ise bayt koda dönüştürüldükten sonra koşturulmaktadır. Fakat, JavaScript haricindeki dillere karşılık bir uygulayıcı kütüphaneyi projenize eklemeniz gerekmektedir.

Örneğin JVM üzerinde Ruby dili ile uygulama geliştirmek istiyoranız, JRuby bağımlılığını Java projenize eklemelisiniz.

JRuby Maven Dependency.

```
<dependency>
    <groupId>org.jruby</groupId>
    <artifactId>jruby</artifactId>
```

```
    <version>1.7.16</version>
</dependency>
```

Diğer listeli diller için de benzer biçimde gereken bağımlılık Java projenize eklenmelidir.

Fakat, JavaScript programlama dili için olay biraz farklı bir durumda. Çünkü, Java 7 Rhino, Java 8 ise Nashorn isimli JavaScript motorlarını gömülü olarak JVM içerisinde bulundurmaktadır. Bu Java ekosisteminin JavaScript diline ne kadar önem verdiğini ayrıca göstermektedir.

Java Scripting API

Java programlama dili, tüm bu listeli dilleri koşturabilmek için ortak arayüzlerin bulunduğu bir API sunmaktadır. Java Scripting API bileşenleri `javax.script` paketi içerisinde bulunmaktadır.

`javax.script` paketi oldukça basit arayüz ve sınıflar içermektedir.Bunlardan en önemlisi `ScriptEngine` arayüzüdür.

ScriptEngine

ScriptEngine türünden nesneler, `ScriptEngineManager#getEngineByName` metodu üzerinden eşsiz bir takma isim ile elde edilmektedir. Bu nesneler ile, String türünden kod blokları koşturulabilmekte, ayrıca Java ile iletişim kurulabilmektedir. Örneğin, Nashorn JavaScript motoru için `"nashorn"` veya `"rhino"` takma adları, Ruby için ise `"jruby"` takma adı kullanılmaktadır.

Örneğin;

```
...
ScriptEngineManager engineManager = new ScriptEngineManager();

ScriptEngine engine = engineManager.getEngineByName("nashorn");
❶
ScriptEngine engine = engineManager.getEngineByName("rhino"); ❷
ScriptEngine engine = engineManager.getEngineByName("jruby"); ❸
ScriptEngine engine = engineManager.getEngineByName("jython");
❹
...
```

❶ Java 8 için JavaScript motoru

❷ Java 7 için JavaScript motoru

❸ Ruby için JRuby motoru

❹ Python için Jython motoru

Nashorn JavaScript Motoru

Nashorn, Java 8 için özel olarak sıfırdan geliştirilen bir JavaScript motorudur. Nashorn, Rhino JavaScript motoruna göre 5 kat daha fazla performans sunmaktadır.

Nashorn JavaScript motoru `EcmaScript 5` standardını desteklemekte ve tüm testlerini başarıyla geçmiş bulunmaktadır.

JVM dillerinden Java Scripting API destekleyenler, `ScriptEngine#eval` metodu ile kod bloklarını koşturma imkanı elde etmektedir. Bu sayede ortak arayüz bileşenleri üzerinden Java harici diller JVM üzerinde koşturulabilmektedir.

Nashorn Engine Örneği.

```
ScriptEngineManager engineManager = new ScriptEngineManager();

ScriptEngine engine = engineManager.getEngineByName("nashorn");
❶

engine.eval("function topla(a,b){ return a + b; }"); ❷
String sonuc=(String)engine.eval(topla(3,5););); ❸
System.out.println(sonuc); // 8
```

❶ Nashorn Engine elde ediliyor.

❷ `topla` isimli JavaScript fonksiyonu tanımlanıyor.

❸ `topla` fonksiyonu Nashorn ile koşturuluyor, ve sonucu elde ediliyor.

Siz de Java Scripting API destekleyen diğer dilleri JVM ekosisteminde koşturabilirsiniz.

Tekrar görüşmek dileğiyle..

Bölüm 11. Java 8 Optional Yeniliği

Bir Java geliştiricisinin korkulu rüyası `NullPointerException` istisnalarıyla uğraşmaktır. `null` değer ile karşılaşmak, ona karşı önlem almak herzaman için can sıkıcı olmuştur. Bu can sıkıcılığı ortadan kaldırmak için Java 8 içerisinde `Optional` sınıfı getirilmiştir. `Optional` yapısı daha evvelden farklı dil ortamlarında bulunan bir özelliktir.

`Optional` türünden nesneler, `null` olma ihtimali olan alanları kolay yönetmek için oluşturulmuştur.

Optional Oluşturmak

Bir `Optional` nesnesi, `Optional` sınıfının çeşitli statik metodlarıyla oluşturulmaktadır. Bunlar `empty`, `of` ve `ofNullable` 'dir.

empty
: Taze bir `Optional` nesnesi oluşturur.

of
: Bir nesneyi `Optinal` ile sarmalar. Parametre olarak `null` değer kabul etmez.

ofNullable
: Bir nesneyi `Optinal` ile sarmalar. Parametre olarak `null` değer kabul eder.

Örneğin

```
Optional<Double> empty = Optional.empty(); ❶
Optional<String> of = Optional.of("Merhaba Dünya"); ❷
Optional<String> ofNull = Optional.of(null); ❸
Optional<Integer> ofNullable = Optional.ofNullable(null); ❹
```

❶ Değer içermeyen Opt

❷ String türünden değer içeren Opt

❸ `Optional#of` `null` kabul etmez. İstisna fırlatır.

❹ `Optional#ofNullable` `null` kabul eder. İstisna fırlatmaz.

#ifPresent - Varsa yap, yoksa yapma

Eğer bir `Optional` içerisinde sadece veri varsa (`null` değilse) bir işin yapılması isteniyorsa `#ifPresent` metodu kullanılabilir. `#ifPresent` metodu `Consumer<T>` fonksiyonel arayüzü türünden bir nesne kabul etmektedir.

Örneğin bir sayının karesini almaya çalışalım. Kullanılan değişken `null` değerini referans ediyorsa `NullPointerException` istisnası alınacaktır.

```
Integer numara = null;

Double karesi = Math.pow(numara , 2); ❶

System.out.println("Sonuç: " + karesi);
```

❶ Exception in thread "main" `java.lang.NullPointerException`

Bu istisna için `if` deyimiyle karşı önlem alınabilir.

```
Integer numara = null;

if(numara != null) {

    Double karesi = Math.pow(numara , 2);

    System.out.println("Sonuç: " + karesi);

}
```

Fakat `if` deyimiyle birlikte `!` , == , `!=` ifadelerini kullanmak akıcı bir geliştirim deneyimi sunmaz. Ayrıca bu durum hata yapılmasına daha açıktır. Bunun yerine `Optional#ifPresent` metodunu kullanabiliriz.

```
Integer numara = null;

Optional<Integer> opt = Optional.ofNullable(numara);

opt.ifPresent(num -> {
```

```
    Double karesi = Math.pow(num , 2);
    System.out.println("Sonuç: " + karesi);
});
```

#map - Dönüştürme

`Optional` nesnelerinin sarmaladığı veriler üzerinde dönüştürüm yapılabilmektedir. Bir önceki örneği bu şekilde yeniden yazabiliriz.

```
Integer numara = null;

Optional<Integer> opt = Optional.ofNullable(numara);

opt
 .map(num->Math.pow(num,2))
 .ifPresent(System.out::println);
```

#filter - Filtreleme

`Optional` nesnelerinin sarmaladığı veriler üzerinde süzme işlemi de yapılabilmektedir.

Örneğin aşağıdaki kod parçası yerine;

```
String message = null;

if (message != null)
    if (message.length() > 5)
        System.out.println(message);
```

Aşağıdaki `Optional` karşılığını kullanabiliriz.

```
String message = null;
Optional<String> opt = Optional.ofNullable(message);

 opt
    .filter(m -> m.length() > 5)
    .ifPresent(System.out::println);
```

#orElse - Varsa al, yoksa bunu al

`orElse` metodu daha çok ternary (`üçlü`) şart ihtiyacı olduğu durumlarda ihtiyaç duyulabilir. Daha akıcı bir geliştirim deneyimi sunar.

numara `null` değilse numarayı döndür, `null` ise `0` döndür.

```
Integer numara = null;

int result = (numara != null) ? numara : 0;
```

Yukarıdaki üçlü şartı `orElse` ile birlikte kullanabiliriz.

```
Integer numara = null;

Optional<Integer> opt = Optional.ofNullable(numara);

int result = opt.orElse(0);
```

#orElseGet - Varsa al, yoksa üret

Bu metod `orElse` metoduna çok benzer, fakat `orElseGet` metod parametresi olarak `Supplier` fonksiyonel arayüzü türünden nesne kabul eder.

```
List<String> names = Arrays.asList("Ali","Veli","Selami");

Optional<List<String>> opt = Optional.ofNullable(names);

names = opt.orElseGet(()-> new ArrayList()); ❶

names = opt.orElseGet(ArrayList::new); ❷
```

❶ Lambda ile

❷ Metod referans ile

#orElseThrow - Varsa al, yoksa fırlat

`Optional` nesnesi bir değeri içeriyorsa (`null` olmayan) o değeri döndürür, `null` ise de sağlanan istisna nesnesini fırlatır. `orElseThrow` metodu `Supplier` türünden bir nesne kabul eder.

```
Integer numara = null;

Optional<Integer> opt = Optional.ofNullable(numara);

int result = opt.orElseThrow(RuntimeException::new); ❶
```

❶ Varsa döndürür, yoksa yeni bir `RuntimeException` istisnası fırlatır.

Java 8 yeniliklerini Java 8 Ebook ile öğrenebilirsiniz.

Tekrar görüşmek dileğiyle.

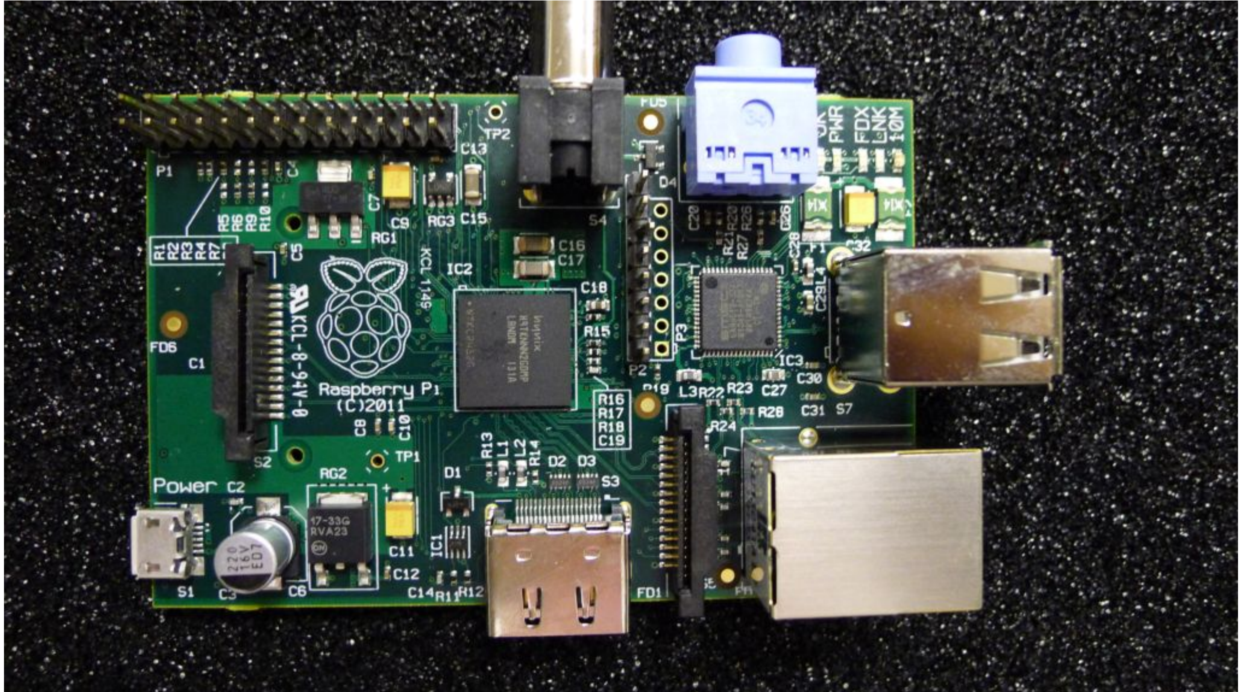
Bölüm 12. Java 8 Embedded

Java 8 Embedded, Java çalışma ortamını (JRE), sınırlı bellek imkanlarına sahip gömülü cihazlarda, az bellek tüketimli olarak sunmayı amaçlayan düşüncenin ürünüdür.

Java çalışma ortamı, farklı işlemci ailesi ve farklı işletim sistemi ailelerine göre ayrıca hazırlandığı için platform bağımsızlığını sunmaktadır. Örneğin bugün `x86` mimarili bir işlemci için `Windows`, `Mac` ve `Linux` türevi işletim sistemlerinde hem çalışma ortamını hem geliştirme ortamını kullanabiliyoruz. Ha keza, `ARM 6/7` işlemci ailesine sahip makinelerde Java çalışma (JRE) ve geliştirme ortamını (JDK) kullanabiliyoruz.

Gömülü sistemlerde `ARM` mimarili işlemciler çok fazla tercih ediliyor. Örneğin elinizdeki akıllı telefonun `ARM` ailesinden bir işlemci olma olasılığı çok çok yüksek. Popülerliği oldukça yüksek olan bir gömülü cihaz Raspberry PI' de `ARM` ailesinden bir işlemci kullanıyor.

Şekil 12.1. Raspberry PI



Gömülü cihazların kendine has sınırlılıkları bulunuyor. Bu sınırlılıkların en başında ise bellek sınırlılığı geliyor. ARM işlemci ailesine göre yapılandırılmış full bir JRE, disk ortamında yaklaşık olarak 47 mb yer tutuyor. 47 MB göze çok gözükmeyebilir, ama, örneğin 64 MB bir belleğe sahip gömülü cihaz için 47 MB çok fazla! İşte tam da bu noktada Java 8 Embedded devreye giriyor.

Java 8 Embedded, Java çalışma ortamını (JRE), gömülü cihazlar için kısmi modülerlik ile boyut olarak düşürmeyi amaçlamaktadır. Bu amaçla Java 8 Embedded geliştirim ihtiyaçlarına göre **3** tip JRE profili sunmaktadır. Bir de full profili katarsak toplamda 4 profil var.

Java 8 Embedded, `compact 1`, `compact 2` ve `compact 3` profillerini sunmaktadır. Bu profillerde, en çok ihtiyaç duyulabilecek spesifik Java paketleri gruplandırılarak boyut bakımından küçülme sağlanmaktadır.

Buna ilaveten, standart bir JRE için iki JVM modu bulunmaktadır. Bunlar `client` ve `server` mode dur. Bu iki seçenekte çalışma ortamına göre JIT Compiler bazında ayrıştırma yapılmaktadır. Java 8 Embedded için ise varsayılan olarak `client` ve `server` modu haricinde `minimal` modu gelmektedir. `minimal` modda bellek tüketimi minimize edilmektedir. Fakat bu modda azami `%5` 'lik bir performans düşümü makul karşılanmaktadır.

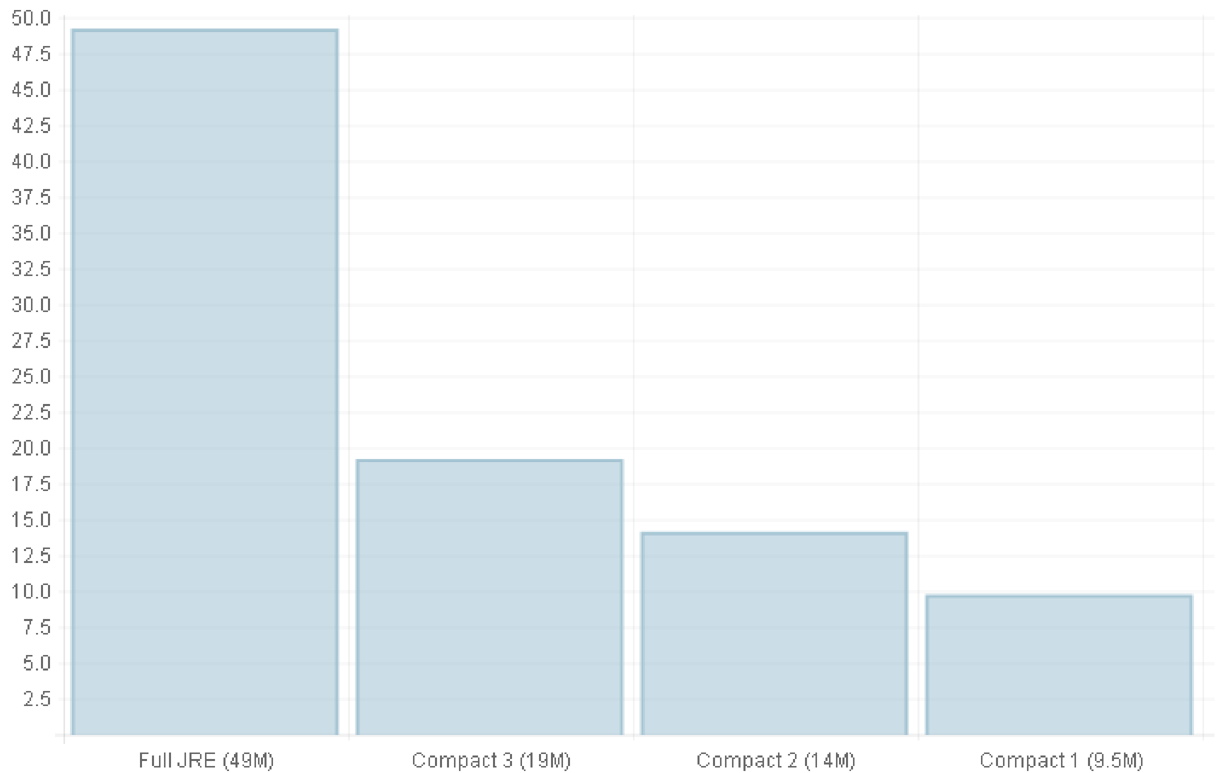
Birbirini içerir biçimde yapılandırılan `Compact` profiller, sık tercih edileceği düşünülen paketler bazında gruplandırılmıştır. Bu gruplamayı aşağıdaki şekilden görebilmekteyiz.

Şekil 12.2. `Full JRE` > `compact 3` > `compact 2` > `compact 1`

Full SE API	Beans	Input Methods	IDL
	Preferences	Accessibility	Print Service
	RMI-IIOP	CORBA	Java 2D
	Sound	Swing	
	AWT	Drag and Drop	
	Image I/O	JAX-WS	
compact3	Security[1]	JMX	JNDI
	XML JAXP[2]	Management	Instrumentation
compact2	JDBC	RMI	XML JAXP
compact1	Core (`java.lang.*`)	Security	Serialization
	Networking	Ref Objects	Regular Expressions
	Date and Time	Input/Output	Collections
	Logging	Concurrency	Reflection
	JAR	ZIP	Versioning
	Internationalization	JNI	Override Mechanism
	Extension Mechanism	Scripting	

Örneğin gömülü sisteminizde en temel Java paketlerini kullanacaksanız `compact 1` profilini seçmek size avantaj sağlayacaktır. `compact 1` profilinde hazırlanan JRE yaklaşık 9.5 MB'dir. Profiller arası boyutsal kıyaslamaya dair grafiği aşağıda görüyoruz.

Şekil 12.3. Compact profile karşılaştırmaları



JavaFX Extension

JavaFX eklentisi kullanıldığında, gömülü cihazınızda JavaFX kullanmak için gerekli paketler Embedded JRE içine dahil edilmektedir. Elbette, oluşan çıktıların boyutları artmaktadır (Yaklaşık 10M kadar daha).

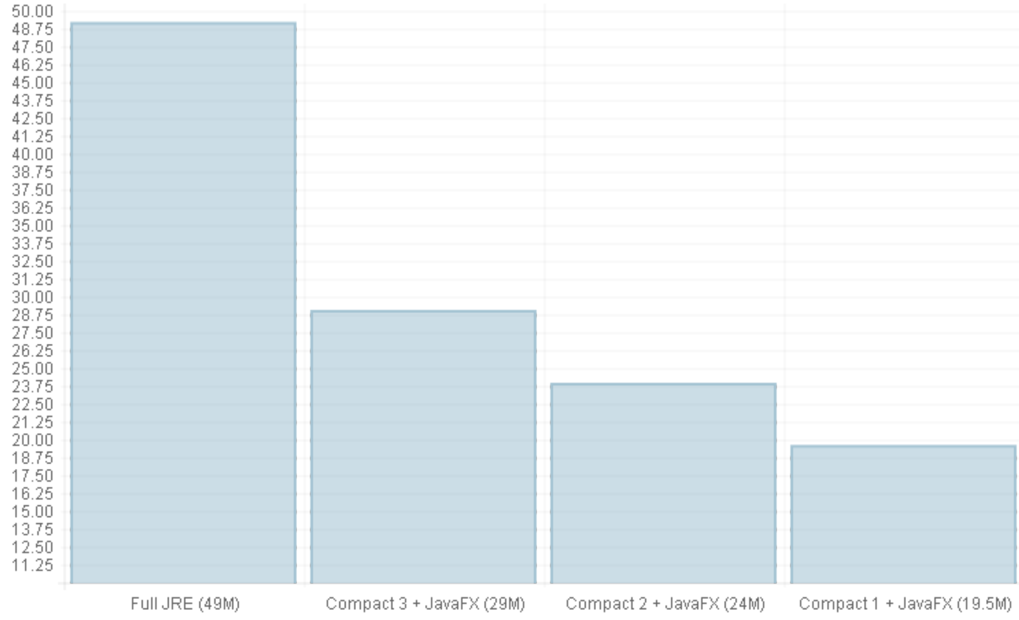
```
> jrecreate -p compact1 -x fx:controls --dest ejdk-compact1-javafx ❶
> jrecreate -p compact2 -x fx:controls --dest ejdk-compact2-javafx ❷
> jrecreate -p compact3 -x fx:controls --dest ejdk-compact3-javafx ❸
```

❶ `ejdk-compact1-javafx` dizininde `compact1` profilli `JavaFX` içeren JRE oluşturur.

❷ `ejdk-compact2-javafx` dizininde `compact2` profilli `JavaFX` içeren JRE oluşturur.

❸ `ejdk-compact3-javafx` dizininde `compact3` profilli `JavaFX` içeren JRE oluşturur.

Şekil 12.4. Compact profile karşılaştırmaları (JavaFX dahil)



Nashorn Extension

Java 8 ile birlikte gelen Nashorn JavaScript motoru, bir eklenti olarak ürettiğiniz `ejre` içine dahil edilebilmektedir. Bu sayee JVM içinde JavaScript dilinde yazılan uygulamaları çalıştırma imkanı elde edilmektedir. Nashorn eklentisi `ejre` çıktısına yaklaşık **`1MB`** ilave etmektedir.

```
> jrecreate -p compact1 -x nashorn --dest ejdk-compact1-nashorn
❶
> jrecreate -p compact2 -x nashorn --dest ejdk-compact2-nashorn
❷
> jrecreate -p compact3 -x nashorn --dest ejdk-compact3-nashorn
❸
```

❶ `ejdk-compact1-nashorn` dizininde `compact1` profilli `Nashorn` içeren JRE oluşturur.

❷ `ejdk-compact2-nashorn` dizininde `compact2` profilli `Nashorn` içeren JRE oluşturur.

❸ `ejdk-compact3-nashorn` dizininde `compact3` profilli `Nashorn` içeren JRE oluşturur.

Not

`-x` parametresiyle JavaFX eklentisi belirtildiğinde, JavaFX üretilen JRE içine dahil edilmektedir. `-x` parametresi `fx:controls`, `sunec`, `sunpkcs11`, `locales`, `charsets`, `nashorn` değerlerini kabul etmektedir. Birden fazlasını aynı anda kullanmak için (`,`) kullanılabilmektedir.

Java 8 ME vs Java 8 Embedded

Java Me ile Java Embedded'in birbirine karıştırılması oldukça olası. Java ME, gömülü cihazlarda Java sanal makinesinin (JVM) çok çok küçük bellek tüketerek çalışmasına olanak sağlayan özel bir Java çalışma ortamıdır. Java 8 Me ile gömülü cihazların donanımsal birimlerini kontrol etmek mümkündür. Örneğin bir gömülü cihazın giriş/çıkış pinlerini, Watchdog Timer gibi bileşenlerini kullanabilirsiniz. Java ME içinde bunları kullanabilmek için özel paket ve sınıflar yer almaktadır.

Tekrar görüşmek dileğiyle.

Bölüm 13. CompletableFuture ile Asenkron Programlama

`CompletableFuture` sınıfı, Java 8 içerisinde asenkron operasyonlar için özelleştirilen bir sınıftır. Java ortamında Java SE ve Java EE teknolojilerinde bir çok asenkron programlama imkanı halihazırda geliştiricilere sunulmaktadır. CompletableFuture sınıfı ise, asenkron programla ihtiyaçlarına çok daha genel çözümler getirmektedir.

Syncronous vs. Asyncronous

Eğer bir uygulamanın akışında, bir görevin başlaması diğer görevin bitişine bağlı ise, buna senkron programlama; Eğer bir görevin başlaması diğer görevin başlamasına engel olmuyorsa da asenkron programlama kavramları ortaya çıkmaktadır. Java programlama dili asenkron programlamaya çoğu noktada imkan sağlamakla birlikte, dilin genel yatkınlığı çoğu dil gibi senkron programlama yönündedir. Fakat, örneğin JavaScript gibi bir dili incelediğinizde, asenkronitinin dilin diyaznını ne derece etkilediğini gözlemleyebilirsiniz.

Örneğin, elimizde `fetchFromDatabase` ve `saveFiles` metodları olduğunu varsayalım. İlk metodun koşturulma süresi 5, diğerinin ise 3 saniye alıyor olsun.

```
private List<String> fetchFromDatabase(){
   ...
       Thread.sleep(5000);
   ...
}

private List<byte[]> readFiles(){
   ...
       Thread.sleep(3000);
   ...

}
```

Şimdi bu iki metodu peşisıra koşturalım.

```
fetchFromDatabase();
readFiles();
```

Bu iki görevin tamamlanma süresi ne kadar olacak? **Cevap**: `Math.sum(5,3) = 8`

Java dilinin genel doğası gereği bu iki iş sırasıyla işletilecektir. Fakat dikkat edilirse, yapılan iki iş birbirinden tamamen bağımsızdır. Biri DB'den veri çekiyor, diğeri ise dosyalama sisteminden dosya okuyor. Dolayısıyla, bu işlerden birinin başlaması için diğer işin tamamlanması beklenmek zorunda değil.

Bu iki metodun asenkron olarak çalışması için geleneksel çokişlemcikli programlama ile harici asenkron iş kolları oluşturulabilir. Fakat, burada geleneksel yöntemlerin dışında `CompletableFuture` nesnesi üzerinden gitmekte fayda görüyorum.

```
public class CompletableFuture<T> implements Future<T>,
CompletionStage<T> {
    ...
}
```

`CompletableFuture` sınıfı `Future` ve `CompletionStage` arayüzleri türünden jenerik bir sınıf. CompletableFuture türünden nesneler, nesnenin yapılandırıcısı üzerinden veya `CompletableFuture` 'nin çeşitli statik metodlarıyla oluşturulabilmektedir.

`CompletableFuture` ile doğası senkron koşmak olan bir işi, asenkron koşar hale getirebilirsiniz. Aslında yapılan iş, senkron koşan işin arka plana itilerek koşturulması ve mevcut program akışının kesintiye uğratılmamasıdır. `CompletableFuture` nesneleri, ekstra olarak tanımlanmadığı sürece tek bir ForkJoin Thread havuzu ile işlerini asenkron olarak arka planda koşturmaktadır.

Şimdi yukarıdaki senkron örneği asenkron hale getirelim. Bunun için CompletableFuture#runAsync metodu kullanılabilir.

```
public static CompletableFuture<Void> runAsync(Runnable
runnable) {
        ...
   return f;
}
```

`CompletableFuture#runAsync` metodu `Runnable` türünden bir görev sınıfı kabul etmektedir, arından CompletableFuture türünden bir nesne döndürmektedir. Parametre olarak iletilen Runnable nesnesi, arkaplanda asenkron olarak koşturulmaktadır.

Not

Runnable arayüzü tek bir soyut metoda sahip olduğu için, Lambda fonksiyonu olarak temsil edilebilir. `() → { }`

```
CompletableFuture<Void> futured1 =
CompletableFuture.runAsync(() -> {

        fetchFromDatabase(); ❶

});

CompletableFuture<Void> futured2 =
CompletableFuture.runAsync(() -> {

        saveToFile(); ❷

});

futured1.join(); ❸
futured2.join(); ❹
```

Yukarıdaki **(1)** ve **(2)** numaralı işler bu noktadan sonra arkaplanda ForkJoin thread havuzu içinde koşturulmuş olacak. Böylece **(2)** numaralı iş, **(1)** numaralı iş koşturulmaya başlatıldıktan hemen sonra çalışmaya başlayacak, diğerinin işe koyulmasını bloke etmeyecek.

Peki şimdi bu iki asenkron görevin tamamlanma süresi ne kadar olacak? **Cevap**: `Math.max(5,3) = 5`

Burada iki iş birden hemen hemen aynı anda başlayacağı için, iki işin toplamda tamamlanma süresi yaklaşık olarak en fazla süren görev kadar olacaktır.

Not

`CompletableFuture#join` metodu, asenkron olarak koşturulan görev tamamlanana kadar, uygulama akışının mevcut satırda askıda kalmasını sağlar. Yani **(3)** ve **(4)** satırlarından sonraki satırlarda, yukarıdaki iki işin birden tamamlanmış olduğunu garanti edebiliriz.

CompletableFuture#allOf

Birden fazla CompletableFuture nesnesini birleştirir. Ancak herbir iş birden tamamlandığında, CompletableFuture nesnesi tamamlandı bilgisine sahip olur.

```
public static CompletableFuture<Void> allOf(CompletableFuture<?
>... cfs) {

    ...

}
```

Örneğin;

```
CompletableFuture<Void> future1 = CompletableFuture.runAsync(()
-> {
    ...
        Thread.sleep(5000);
    ...

    System.out.println("İlk görev tamamlandı..");
});

CompletableFuture<Void> future2 = CompletableFuture.runAsync(()
-> {
    ...
        Thread.sleep(15000);
    ...

    System.out.println("Diğer görev tamamlandı..");
});

CompletableFuture<Void> allOf =
CompletableFuture.allOf(future1, future2);

System.out.println("Bir arada iki derede.");

allOf.join();

System.out.println("Bitti.");
```

Yukarıda iki tane asenkron iş koşturulmaktadır. Bir tanesi 5, diğeri ise 15 saniye sürmektedir. Eğer asenkron koşan uygulama akışında, bu iki iş bitene kadar bir noktada beklemek istiyorsak, `CompletableFuture#allOf` dan faydalanabiliriz. Uygulama akışının askıda bekletilmesi ise `CompletableFuture#join` metodu ile sağlanmaktadır.

Çıktı.

```
Bir arada iki derede. // 0. saniyede
İlk görev tamamlandı.. // 5. saniyede
Diğer görev tamamlandı.. // 15. saniyede
Bitti. // 15. saniyede
```

CompletableFuture#anyOf

Birden fazla CompletableFuture nesnesini birleştirir. Herhangi bir görev tamamlandığında, CompletableFuture nesnesi tamamlandı bilgisine sahip olur.

Örneğin;

```
CompletableFuture<Void> future1 = CompletableFuture.runAsync(()
-> {
    ...
        Thread.sleep(5000);
    ...

    System.out.println("İlk görev tamamlandı..");
});

CompletableFuture<Void> future2 = CompletableFuture.runAsync(()
-> {
    ...
        Thread.sleep(15000);
    ...

    System.out.println("Diğer görev tamamlandı..");
});

CompletableFuture<Void> anyOf =
CompletableFuture.anyOf(future1, future2);

System.out.println("Bir arada iki derede.");

anyOf.join();

System.out.println("Bitti.");
```

Çıktı.

```
Bir arada iki derede. // 0. saniyede
İlk görev tamamlandı.. // 5. saniyede
Bitti. // 5. saniyede
Diğer görev tamamlandı.. // 15. saniyede
```

CompletableFuture#supplyAsync

`CompletableFuture#supplyAsync` metodu `CompletableFuture#runAsync` metodu gibidir. Fakat koşma sonucunda geriye bir sonuç döndürebilmektedir. Bir iş sonunda geriye hesaplanmış bir değer döndürmeye ihtiyaç duyulduğu noktada kullanılabilir.

Örneğin, `/var/log` dizinindeki tüm dosya ve klasörlerin listesini hesaplatmak istiyoruz diyelim.

```
CompletableFuture<List<Path>> future =
CompletableFuture.supplyAsync(() -> {
    Stream<Path> list = Stream.of();

    try {
        list = Files.list(Paths.get("/var/log"));
    } catch (IOException e) {
        e.printStackTrace();
    }

    return list.collect(Collectors.toList());

});
```

Bu ihtiyacı `Files#list` metodu ile sağlayabiliriz. `Files#list` metodu tanımlanan dizindeki tüm dizin ve dosyaları bir Path listesi olarak sunmaktadır. Dizindeki dosya ve dizin sayısına göre bir sonucun elde edilmesi belirli bir zaman gerektirebilir.

Not

CompletableFuture#supplyAsync metodu Supplier türünden bir nesne kabul ettiği için bir Lambda fonksiyonu olarak temsil edilebilirdir. `() → T`

CompletableFuture'in çoğu metodu işlerini asenkron olarak arkaplanda koşturmaktadır. Bu sebeple mevcut uygulamanın akışını askıda bırakmamaktadır.

Bir CompletableFuture’in iş bitimindeki sonucunu elde etmenin iki yöntemi bulunmaktadır.

İlk yol, `join()` metodu kullanmak

join() metodu, asenkron olarak işletilen görev tamamlanana kadar uygulama akışını askıda tutmaktadır. İş tamamlandığında ise varsa sonuç değerini döndürmektedir.

```
CompletableFuture<List<Path>> future =
CompletableFuture.supplyAsync(() -> {
    Stream<Path> list = Stream.of();

    try {
        list = Files.list(Paths.get("/var/log"));
    } catch (IOException e) {
        e.printStackTrace();
    }

    return list.collect(Collectors.toList());

});


// Varsa diğer işler bu arada yapılabilir


List<Path> liste = future.join(); ❶


// join() tamamlanana kadar buraya erişim devam etmez
```

❶ İş bitiminde elde edilen sonuç listesi

İkinci yol, `thenAccept*` metodu kullanmak

`thenAccept` metodu ile *callback* stilinde asenkron işlerin sonuçları elde edilebilir. `thenAccept` metodu `Consumer<T>` türünden bir nesne kabul etmekte ve sonucu onun üzerinden sunmaktadır.

```
CompletableFuture<List<Path>> future =
CompletableFuture.supplyAsync(() -> {
    Stream<Path> list = Stream.of();

    try {
        list = Files.list(Paths.get("/var/log"));
    } catch (IOException e) {
        e.printStackTrace();
    }

    return list.collect(Collectors.toList());

});

future.thenAccept( (List<Path> paths) -> {
    // liste burada
});
```

Yukarıdaki `thenAccept` ile, CompletableFuture nesnesine bir hook tanımlanmış olur. İş bitiminde sonuç elde edildiği zaman bu metod otomatik olarak işletilir. Sonuç parametre olarak geliştiriciye sunulur.

CompletableFuture#runAfterBoth*

İki asenkron iş birden tamamlandığında bir `Runnable` türünden nesneyi koşturmayı sağlar.

```
CompletableFuture<Void> future1 = CompletableFuture.runAsync(()
-> {
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});

CompletableFuture<Integer> future2 =
CompletableFuture.supplyAsync(() -> {
    return 10;
});

future1.runAfterBoth(future2,()->{
    System.out.println("İkisi birden bitti"); // 5. saniyede
});
```

CompletableFuture#runAfterEither*

İki asenkron işden herhangi biri tamamlandığında bir `Runnable` türünden nesneyi koşturmayı sağlar.

```
CompletableFuture<Void> future1 = CompletableFuture.runAsync(()
-> {
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
});

CompletableFuture<Integer> future2 =
CompletableFuture.supplyAsync(() -> {
    return 10;
});

future1.runAfterEither(future2,()->{
    System.out.println("İkisinden biri tamamlandı.."); // 0.
saniyede
});
```

CompletableFuture#handle*

`CompletableFuture#handleAsync` metodu bir önceki asenkron görevin sonucunu işlemek ve ardındaki görevlere paslamak için yapılandırılmıştır. `CompletableFuture#handleAsync` ile, birbirini besleyen zincirler şeklinde asenkron iş akışları yazılabilir.

Örneğin, iki asenkron işten birini, diğerini besler şeklinde yapılandıralım.

Görev 1
Asenkron olarak bir dizindeki tüm dosya ve dizinler bulunsun

Görev 2
Bulunan dizinlerin boyut bilgisi asenkron olarak hesaplansın

Görev 3
Dosya yolu ve boyut bilgisi asenkron olarak listelensin.

```
CompletableFuture.supplyAsync(() -> { ❶

Stream<Path> list = Stream.of();

try {
    list = Files.list(Paths.get("/var/log"));
} catch (IOException e) {
    throw new RuntimeException(e);
}

return list.collect(Collectors.toList());

}).handleAsync((paths, throwable) -> { ❷

Map<Path, Long> pathSizeMap = new HashMap<>();

try {
    for (Path path : paths) {
        long size = Files.size(path);
        pathSizeMap.put(path, size);
    }
} catch (IOException e) {
    throw new RuntimeException(e);
}

return pathSizeMap;
```

```
}).thenAccept(map -> { ❸

for (Map.Entry<Path, Long> entry : map.entrySet()) {
    System.out.printf("%s | %d bytes
%n",entry.getKey(),entry.getValue());
}

});
```

❶ Dosya ve dizinleri liste olarak döndürür

❷ Elde ettiği listeden her bir dizinin boyutunu hesaplar, bir Map nesnesi olarak sunar.

❸ En son üretilen Map nesnesinden dosya yolu ve boyutunu birbir çıktılar.

`CompletableFuture` sınıfının Java'da asenkron programlamayı hiç olmadığı kadar kolaylaştırdığını söyleyebilirim.

Tekrar görüşmek dileğiyle.

Bölüm 14. Java ME 8 Embedded ve Raspberry PI

Gömülü sistem teknolojilerinde hiç şüphesiz `Raspberry PI` önemli bir çığır açtı. Boyutunun küçük ama özelliklerinin çok olduğu bu küçük elektronik board, gömülü sistemlere olan ilgi ve alakayı da arttırdı.

Gömülü sistemler tarafında böyle bir durum var iken, Java tarafında da Java programlama dili ve ortamının 8. versiyonu `Java SE 8` piyasaya sürüldü. `Java ME` (Mobile Edition) ise, uzunca bir süredir suskundu. Fakat geçtiğimiz günlerde Java ME 8 sürümü Oracle tarafında yayınlandı. (Bkz. java-me-8-released) Bu sürüm ile artık, Raspberry PI gibi çeşitli gömülü sistem aygıtlarında Java ME 8 kullanılabilmektedir.

Şu anda Java ME 8'in desteklediği platformlar aşağıdaki gibidir;

- Raspberry Pi Model B on Debian Linux
- Freescale FRDM-K64F on mbed
- Qualcomm IoE platform based on QSC6270T and Brew MP

Java ME 8, gömülü cihazlarda düşük bellek tüketimi ve gömülü sistemlere odaklı geliştirilmi çeşitli API'leri ile karşımıza çıkıyor. Bunlar;

- `JSON` API
- Async `HTTP` API
- `OAuth` 2.0 API
- JSR 75 (`File` Connection API)
- JSR 120 (`Wireless` Messaging API)
- JSR 172 (`Web Services` API)
- JSR 177 (Security and Trust Services API)
- JSR 179 (Location API)
- JSR 280 (`XML` API)

Ayrıca Java ME ile çalıştığınız gömülü sistemin donanımsal birimlerini yönetebiliyoruz. Örneğin; `GPIO`, `I2C`, `SPI`, `UART`

Java ME 8 ile Raspberry PI üzerinde LED yakma

Bir programlama dili öğretilirken genelde ilk adım `Hello World` ifadesini konsola çıktılamaktır. Gömülü sistemlerin `Hello World` 'ü ise, bir LED'i yakmaktır. Biraz daha detaylandırırsak, cihazın GPIO (Genel amaçlı giriş çıkış) pinlerini yönetmektir.

Gömülü sistemlerde genel olarak bir pin, hem giriş hem de çıkış olarak kullanılabilmektedir. Şahsım `Raspberry PI Model B Revision 1` sahibiyim ve onda 8 adet `GPIO` pini bulunuyor. Bu pinleri hem giriş, hem de çıkış olarak kullanabilmekteyiz.

Donanımsal Gereksinimler

Yapacağımız uygulamada, merkezde Raspberry PI olmak üzere çeşitli donanımsal araç-gerece ihtiyacımız olacak. Bunları aşağıdaki gibi detaylandırabiliriz;

Raspberry PI

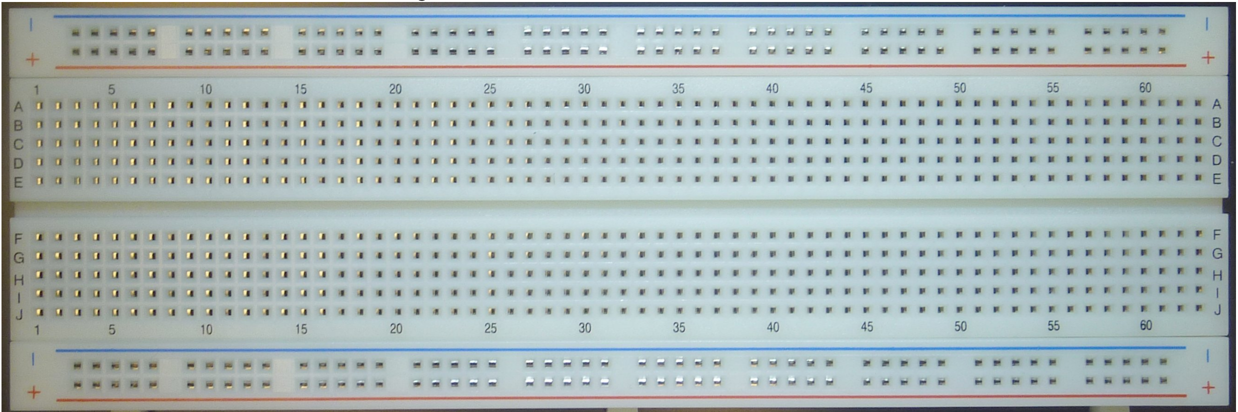
Raspberry PI Model B (Revision 1 veya Revision 2). Bende eski olan Revision 1 bulunuyor. Revision 2'de GPIO pin sayısının daha fazla olduğunu ve birkaç ufak değişikliğin olduğunu görüyorum (Bkz. upcoming-board-revision).

Henüz Raspberry PI'niz yok ise, Farnell veya RS Components gibi firmalardan elde edebilirsiniz. 75 € gümrük sınırını aşmadığı için en uygun elde etme biçimi bu. Fakat robotistan gibi yerli girişimlerden elde etmekte mümkün.

Bread Board

Bread board, elektronik devreleri kolay bir biçimde kurabilmek ve test edebilmek için üretilmiş basit ama önemli bir araç.

Şekil 14.1. Bread Board



Aslında olay çok basit, kenarlarda ikişer adet boylu boyunca uzanan 4 hat bulunyor. Ortada ise, ikiye bölünmüş enlemesine uzanan hatlar var. Hatlar

üzerindeki noktalarda ise, kablo, pin, led, buton gibi bileşenleri yerleştirip, çıkarabileceğiniz delikler bulunuyor. Yani içerisi şöyle;

Şekil 14.2. Bread Board hatları

SD Kart

Raspberry PI'yi satın aldığımda SD Kartı yanında gelmişti, fakat şuan bu uygulama yapılıyor mu bilmiyorum. İşletim sisteminin kurulması için SD kart gerekmektedir.

Adaptör

5V çıkışlı microusb adaptör tercih edebilirsiniz.

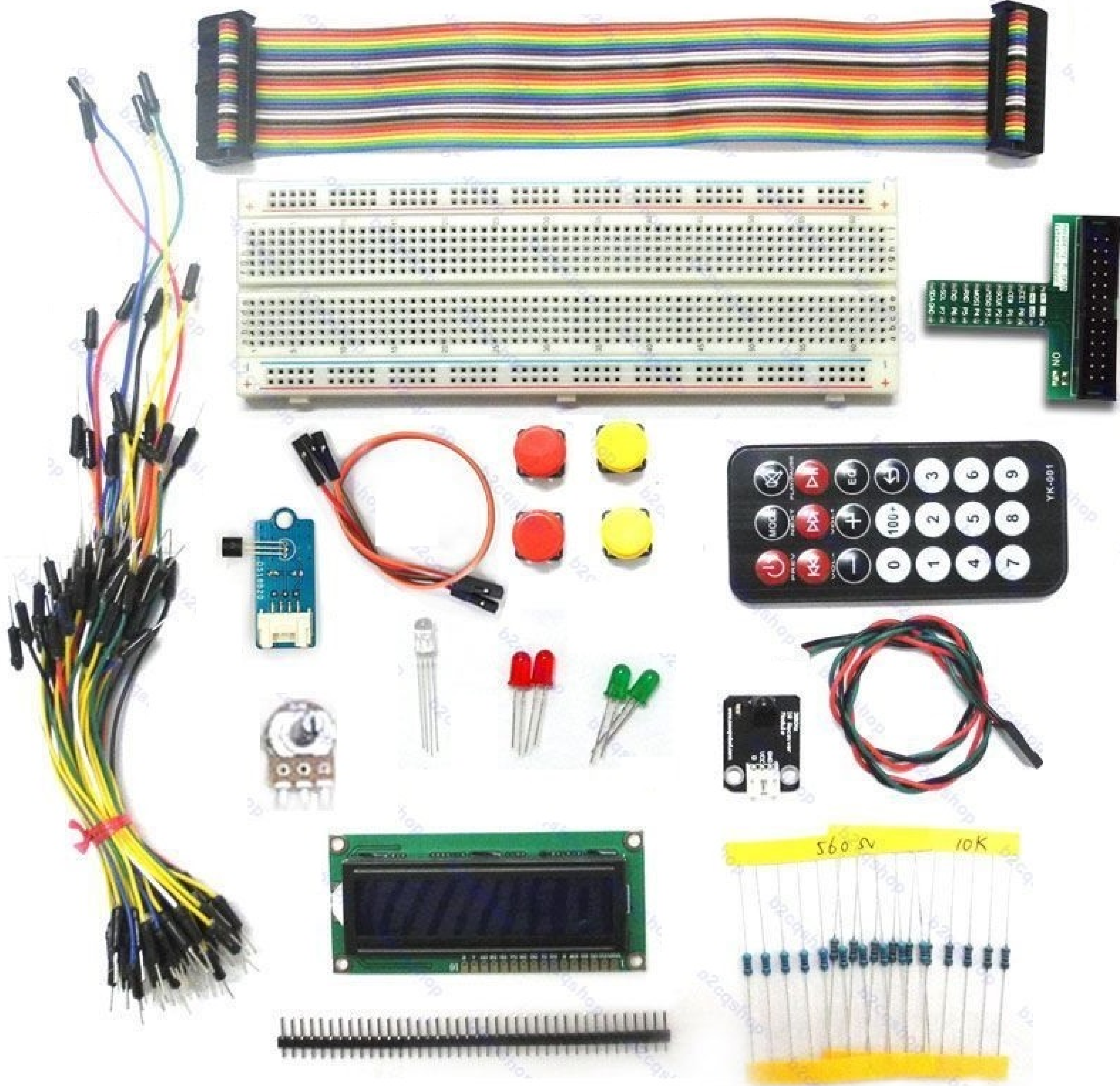
Wifi adapter

Raspberry PI'de ethernet girişi bulunuyor fakat, çalışma ortamınızda kablonuz yok ise bir mini wifi bileşeni almak en iyi çözüm olacaktır.



Kablo, Led, GPIO Extension Board

Bu gereçleri Ebay üzerinden toplu olarak elde etmiş idim. Raspberry PI Starter Kit



Not

Ebay ile uzakdoğu ülkeleri üzerinden verilen siparişler yaklaşık 15-30 gün arasında ulaşmaktadır.

HDMI Kablosu

Raspberry PI üzerinde HDMI çıkışı bulunmaktadır. Ben HDMI destekli bir televizyon kullandım. Bu maksatla 2 metrelik bir HDMI kablosu almayı tercih ettim.

Klavye ve Maus

USB destekli bir klavye ve maus olmalıdır.

Yazılımsal Gereksinimler

Java ME 8 ile başlarken Netbeans veya Eclipse IDE'lerini kullanabiliriz. Bir anti-eclipser olarak Netbeans kullanmayı tercih ettim :)

Java ME 8 SDK

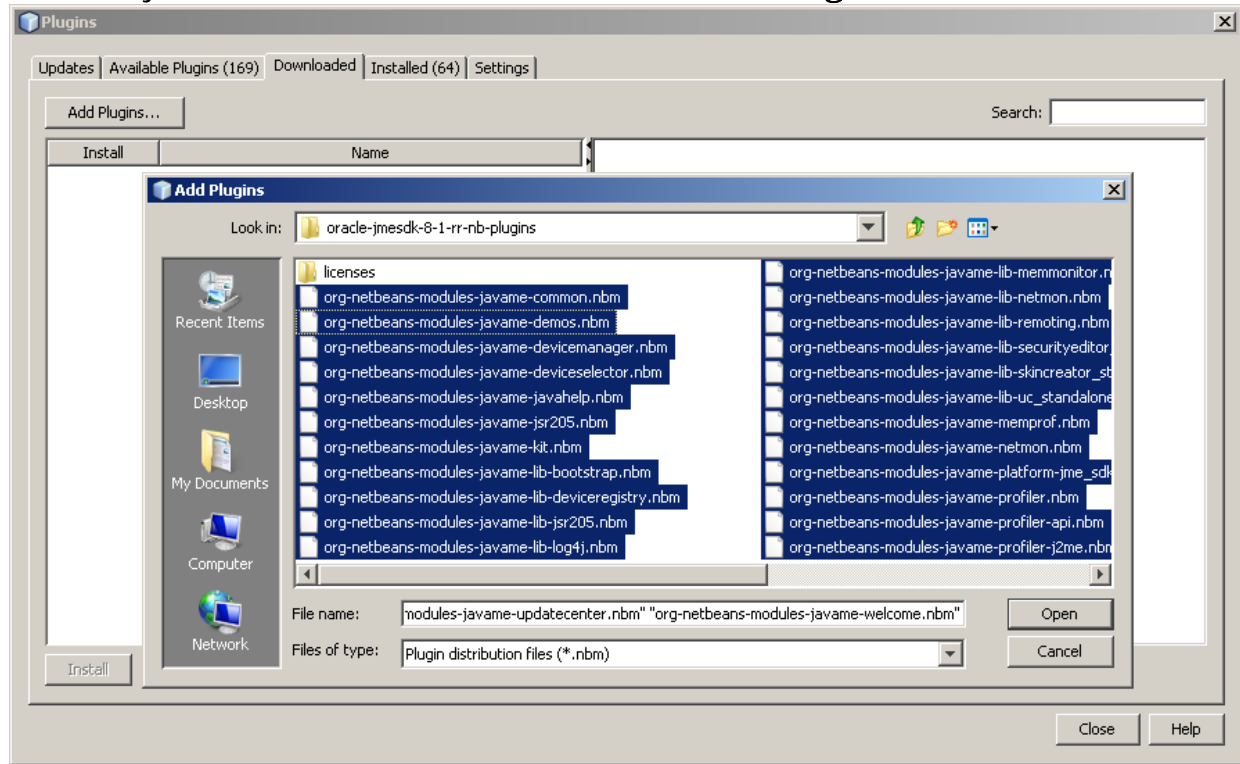
Java ME 8 SDK'yi şuanlık sadece Windows ortamında kullanabiliyoruz. Java ME SDK 'yı bu adresten edinerek sistemimize kurabiliriz.

Java ME 8 Plugin (Netbeans veya Eclipse)

Java ME SDK Plugins sayfasından `Java ME SDK Plugins for NetBeans` indiriyoruz. İndirdiğimiz, zip dosyasını bir istediğimiz bir dizine ayıklayıp, içindeki .npm uzantılı eklentilerin hepsini yüklüyoruz.

`Tools > Plugins > Downloaded`

Şekil 14.3. Java ME 8 SDK Netbeans Pluginin Kurulması



Java ME 8 Embedded

Raspberry PI üzerine kurmak üzere yaklaşık 3.5 MB’lik Java ME 8 Embedded' indiriyoruz. (**Not**: Birazdan kurulumundan bahsedeceğiz.)

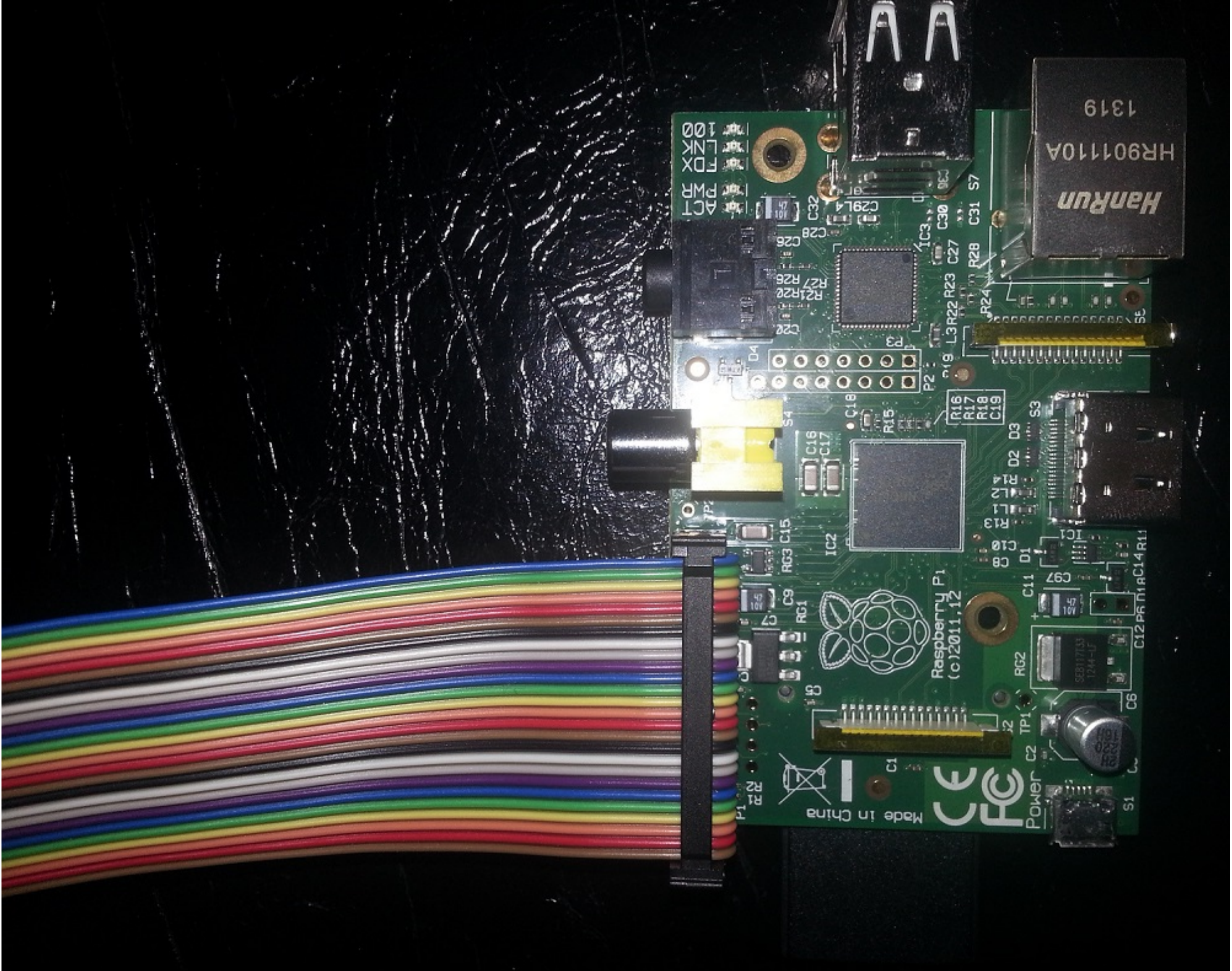
Raspbian OS Kurulması

İlk iş olarak, Raspberry PI SD kartı üzerine, Debian tabanlı bir işletim sistemi (Raspian) kurulmalıdır. http://www.raspberrypi.org/downloads/ sayfasından indirilerek talimatları ile kurulmalıdır. Sonrasında ise, network bağlantısı sağlanmalıdır.

Devrenin Kurulması

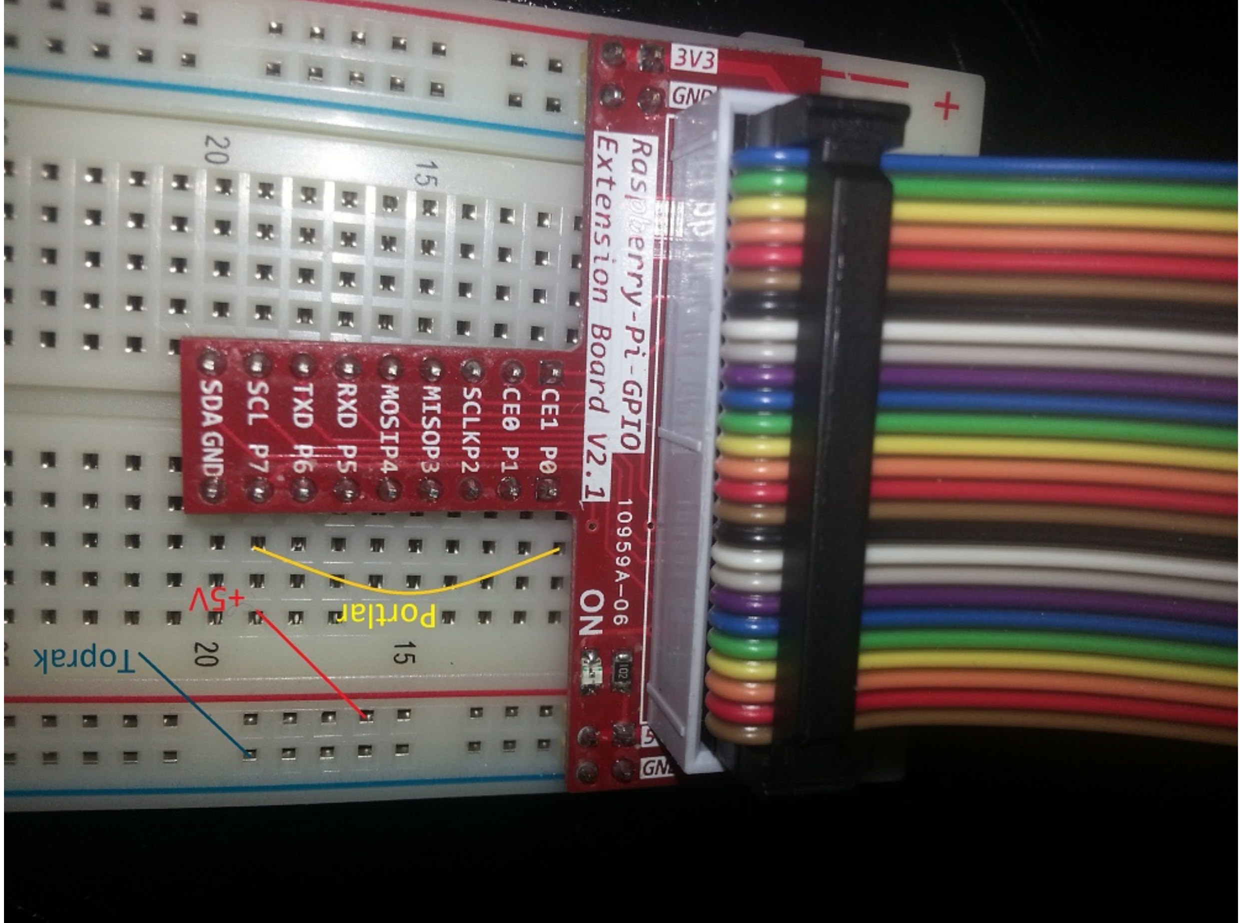
Raspberry PI'nin tüm portlarını bread board ile entegre etmek için, GPIO extension kiti ve ara kablosu kullanmak kolaylık sağlayacaktır.

Şekil 14.4. GPIO Extension kablosunun Raspi'ye takılması



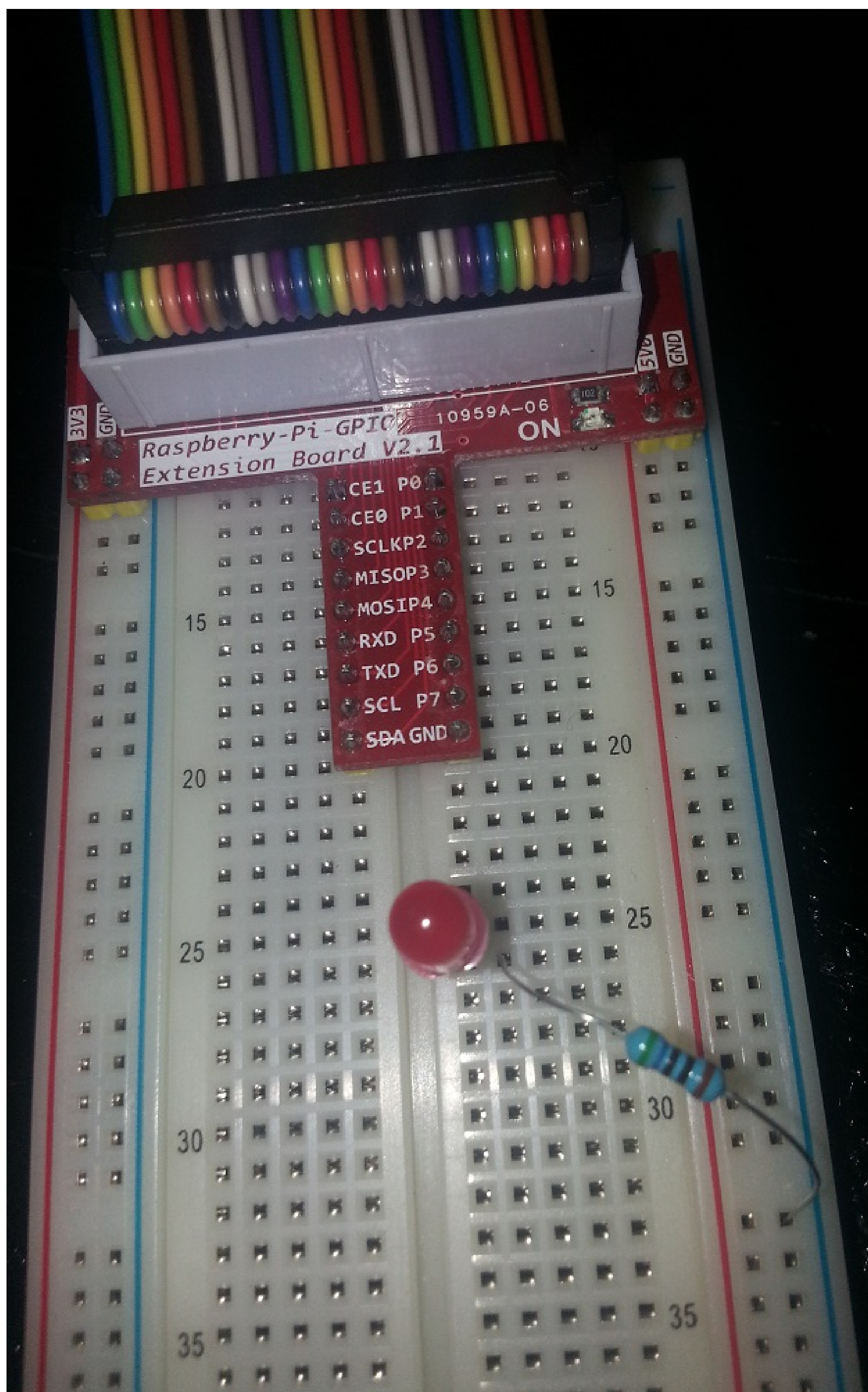
Sonrasında ise GPIO extension kiti kablonun öteki ucuyla beraber bread boarda takılmalıdır.

Şekil 14.5. GPIO Extension’in bread boarda takılması



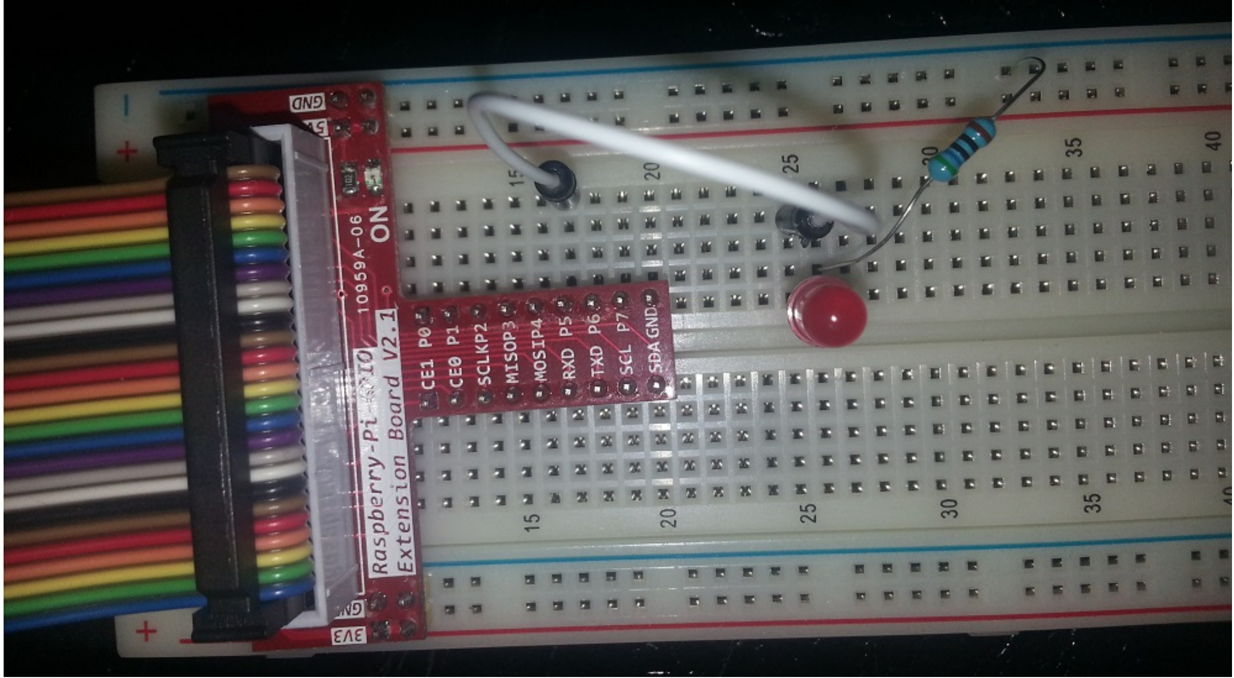
Sonrasında, bir LEDin katot (-) ucu bir direnç ile birlikte, bread board üzerindeki mavi (toprak) hattına bağlanmalıdır. Burada Starter Kit içinden çıkan 560 ohm luk direnci kullandık.

Şekil 14.6. Direnç ve LEDin takılması


3V3
GND
Raspberry-Pi-GPIO
Extension Board V2.1
10959A-06
ON
5V0
GND
CE1 P0
CE0 P1
SCLKP2
MISOP3
MOSIP4
RXD P5
TXD P6
SCL P7
SDA GND
15
20
25
30
35

Bir sonraki aşamada ise LEDin anod (+) ucuyla, P0-P7 arası portlardan herhangi birisini, bir kablo ile birleştiriyoruz. Bu uygulamada P5 portunu tercih ettin.

Şekil 14.7. LED ve portun birbirine bağlanması



Bu aşamada devrenin kurulması tamamlanmış oluyor.

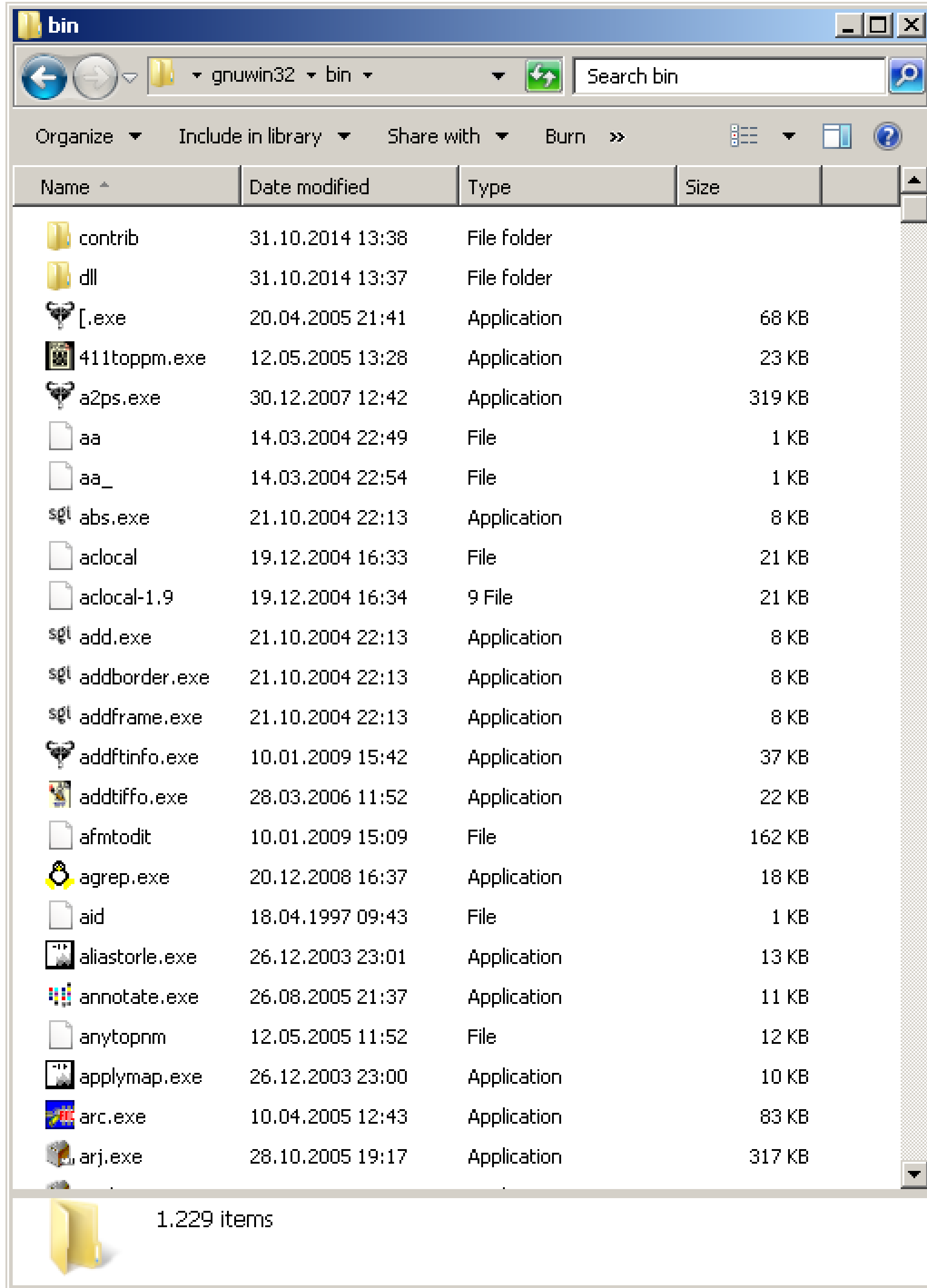
Java ME 8'in Raspberry PI'ye Yüklenmesi

Raspberry PI'ye bir Network bağlantısı ile ilişkilendirdikten sonra, çalıştığımız makineden Raspiye bir SSH bağlantısı kurarak tüm ihtiyaçlarımızı çalışma makinamiden gerçekleştirebiliriz.

Raspberry PI ile SSH bağlantısı kurarken ve Raspberry PI'ye Java ME 8'i transfer ederken çeşitli Linux araçlarına ihtiyaç duymaktayız. Gereken Linux komutlarını Windows ortamında kullanabilmek için `gnuwin32` projesinden faydalanabiliriz.

- gnuwin32 indirilir.
- İndirilen `.exe` bir dizine ayıklanır.
- Ayıklanılan dizine konsoldan geçilir ve `download.bat` çalıştırılır.
- `download.bat` sonrası `install.bat` çalıştırılır. Belirli bir süre sonra yükleme tamamlanır ve Linux komutlarını taklit eden araçlar ayıklanan `/gnuwin32/bin` dizininde belirir.

Şekil 14.8. Gnuwin32 Linux Araçları



Bu dizin içindeki tüm araçlara, her konsol dizininden erişebilmek için `/bin` dizini Environment Variables (Sistem değişkenleri) ‘a eklenir.

Ardından `gnuwin32` içindeki `scp` aracıyla Java ME’yi Raspiye göndeririz.

```
$ cd /Downloads ❶
$ scp oracle-jmee-8-1-rr-raspberrypi-linux-bin.zip
pi@192.168.2.61:/home/pi/jmee ❷
```

❶ Java ME indirilen dizine geçilir.

❷ `scp` aracıyla Java ME `/home/pi/jmee` dizinine transfer edilir.

Not

`192.168.2.61` adresi, Raspberry PI’nin almış olduğu IP adresidir.

Java ME’nin Çalıştırılması

Raspberry PI’ye gönderilen zip dosyası ayıklanarak, Java ME agent çalıştırılır. Fakat öncesinde SSH ile Raspberry PI’ye bağlantı kurulmalıdır.

```
$ ssh pi@192.168.2.61 ❶
$ pi@192.168.2.61's password: raspberry ❷
```

❶ `192.168.2.61` adresindeki `pi` kullanıcı olarak bağlanılıyor

❷ Şifre giriliyor

SSH ile bağlantı kurduktan sınra Java ME’yi ayıklayıp çalıştırabiliriz.

```
$ cd /home/pi/jmee ❶
$ unzip oracle-jmee-8-1-rr-raspberrypi-linux-bin.zip ❷
$ cd /bin ❸
$ sudo ./usertest.sh ❹
```

❶ `/pi/home/jmee` dizinine geçiliyor.

❷ `zip` dosyası ayıklanıyor.

❸ `/bin` dizinine geçiliyor.

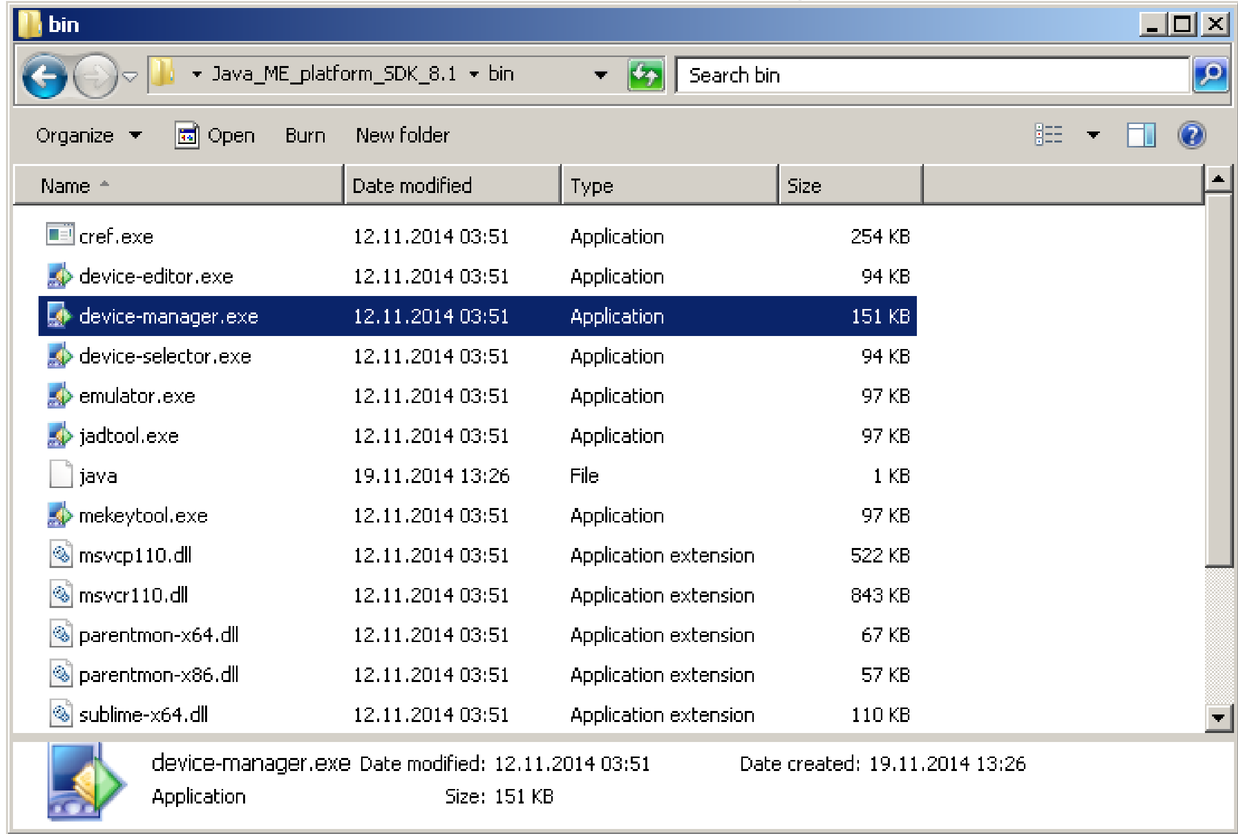
❹ `usertest.sh` scripti `root` yetkisiyle çalıştırılıyor.

Bu aşamada Raspberry PI üzerinde bir Java agent çalışmaya başlıyor. Sonrasında ise Java ME SDK içerisindeki Device Manager ile Raspberry PI’ye uzak bağlantı kurabiliyoruz.

Device Manager’a Rapberry PI Tanıtmak

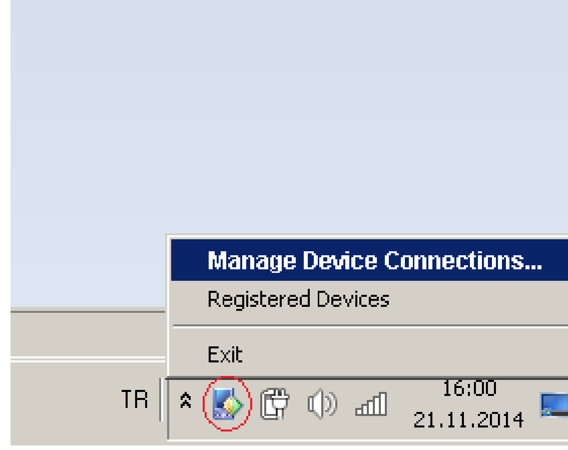
Java ME SDK içinde hem emülator hem de gerçek makineleri yönetebiliyoruz. Biz gerçek bir Raspberry PI cihazını yöneteceğimiz için, Device Manager’a uzaktaki Raspiyi tanıtmalıyız. Device Manager aracı, Java ME SDK yüklediğiniz dizinde `/bin` dizininde yer almaktadır.

Şekil 14.9. Device Manager



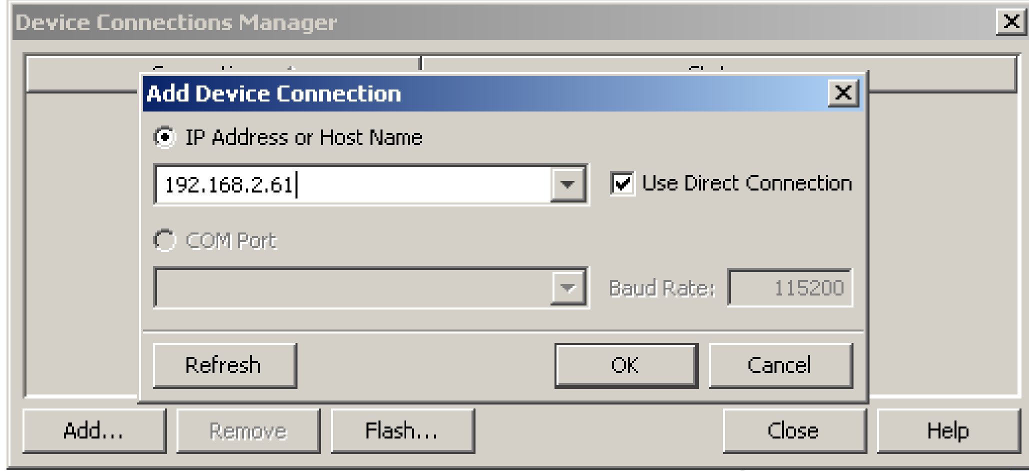
`device-manager.exe` aracını çalıştırarak, uzaktaki Raspberry PI cihazını ona tanıtabiliriz. Küçültülmüş sistem penceresinde bulunan `Device Manager` 'a sağ tuşluyoruz ve `Manage Device Connections` menüsünü seçiyoruz.

Şekil 14.10. Manage Device Connections



Ardından Raspberry PI'nin almış olduğu IP adresini yazarak onu çalışma makinemize tanıtıyoruz.

Şekil 14.11. Add Device Connection



Bu aşamada, Raspberry PI üzerinde Java ME 8'i çalıştırmış ve sistemimize tanıtmış oluyoruz. Şimdi ise uygulamamızı yazıp, Raspberry PI üzerinde çalıştırabiliriz.

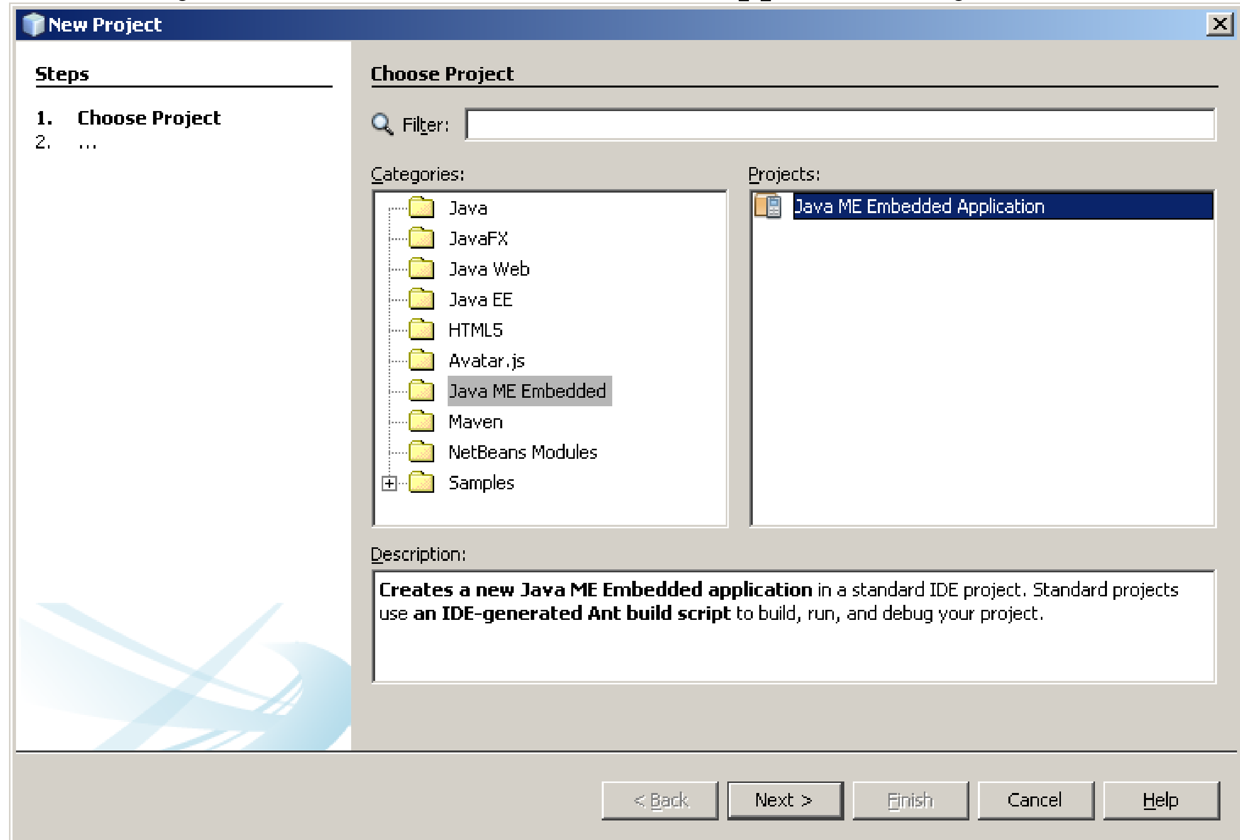
Java ME 8 Embedded Projesinin Oluşturulması

Java ME 8 Embedded projesini Netbeans 8.0.1 ile hazırlayacağız. Bu aşamada Java ME pluginlerinin ve Java ME SDK'nin eksik olmadığından emin olmalıyız.

Adım 1

`File` > `New Project` menüsü takip edilir ve `Java Me Ebdedded` > `Java Me Ebdedded Application` seçilir.

Şekil 14.12. Java Me Ebdedded Application seçilmesi



Adım 2

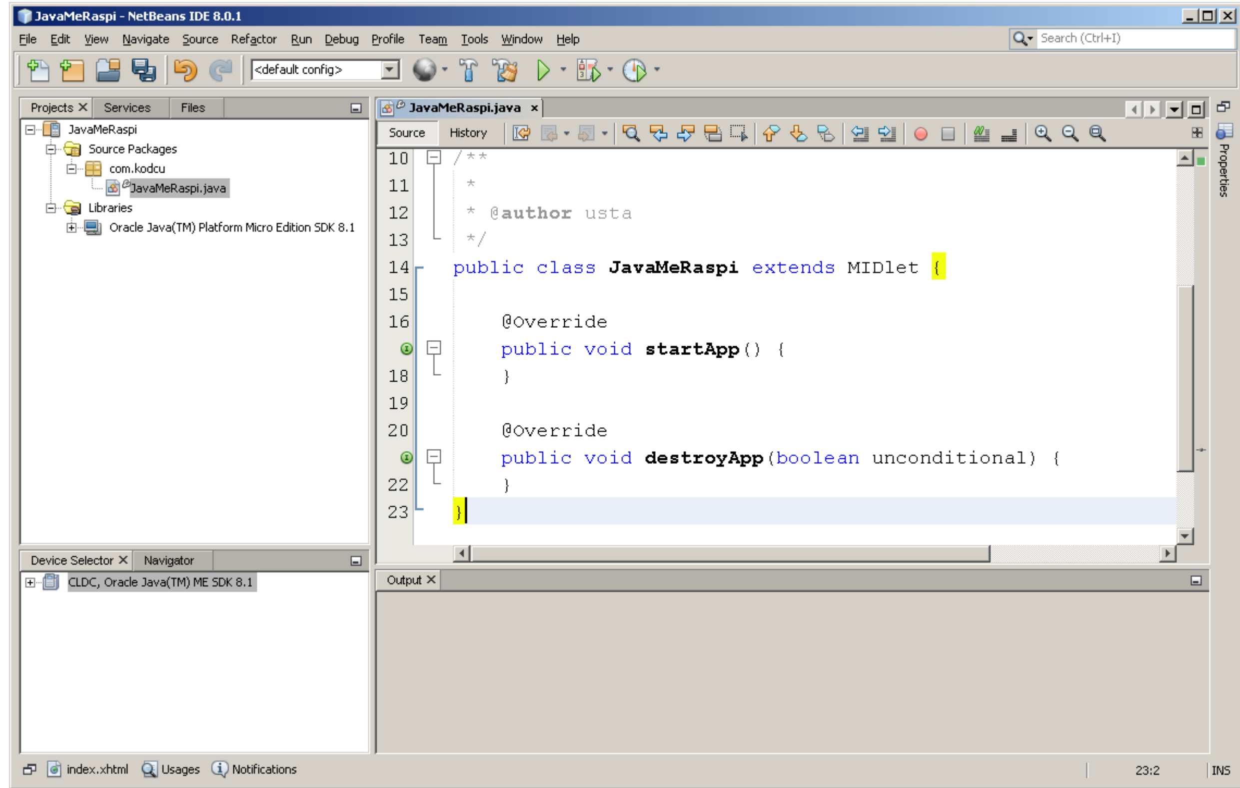
Proje ismi ve gerekliyse diğer kısımlar düzenlenir.

Şekil 14.13. Proje Ayarları

Adım 3

Bu adımda oluşan proje gözlemlenir ve `JavaMeRaspi` adında `Midlet` türünden bir sınıfın oluşturulduğu görülür.

Şekil 14.14. Proje görünümü



Midlet sınıfı içinde `startApp()` ve `destroyApp()` adında iki gövdesiz metod bulunmaktadır. `startApp()` uygulama çalışmaya başladığında, `destroyApp()` ise, uygulama sonlandığında otomatik olarak koşturulmaktadır.

Uygulamanın Yazılması

Artık projeyi oluşturduğumuza göre `JavaMeRaspi` sınıfını düzenleyebiliriz.

```
public class JavaMeRaspi extends MIDlet {

    @Override
    public void startApp() {

        try {

            start(); ❶

        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    private void start() throws IOException {

        GPIOPin led5 = DeviceManager.open(24); ❷

        while (true) { ❸

            led5.setValue(true); ❹
                bekle(1000); ❺
            led5.setValue(false); ❻
                bekle(1000); ❼

        }

    }

    private void bekle(int sure) {
        try {
            Thread.sleep(sure); ❽
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }

    @Override
    public void destroyApp(boolean unconditional) {
        // Uygulama sonlandı
```

```
    }
}
```

❶ `start();` metodu ile akış başlatılıyor.

❷ `Port 5`, nesnel olarak temsil ediliyor.

❸ Sonsuz bir döngü kuruluyor.

❹ `Port 5` enerjilendiriliyor. Led yanıyor.

❺ 1 saniye bekleniyor.

❻ `Port 5` 'in enerjisi kesiliyor.

❼ 1 saniye bekleniyor.

❽ Tanımlanan milisaniye kadar süre bekletiyor.

Bu noktada neden Port 5 için 24 numarasını tercih edildiğini merak edebilirsiniz. Raspberry PI'nin her portu için birer numarası bulunuyor. Bunlar;

Port	**Numara**
GPIO0	17
GPIO1	18

Port	Numara
GPIO2	R1:21 / R2:27
GPIO3	22
GPIO4	23
GPIO5	24
GPIO6	25
GPIO7	4

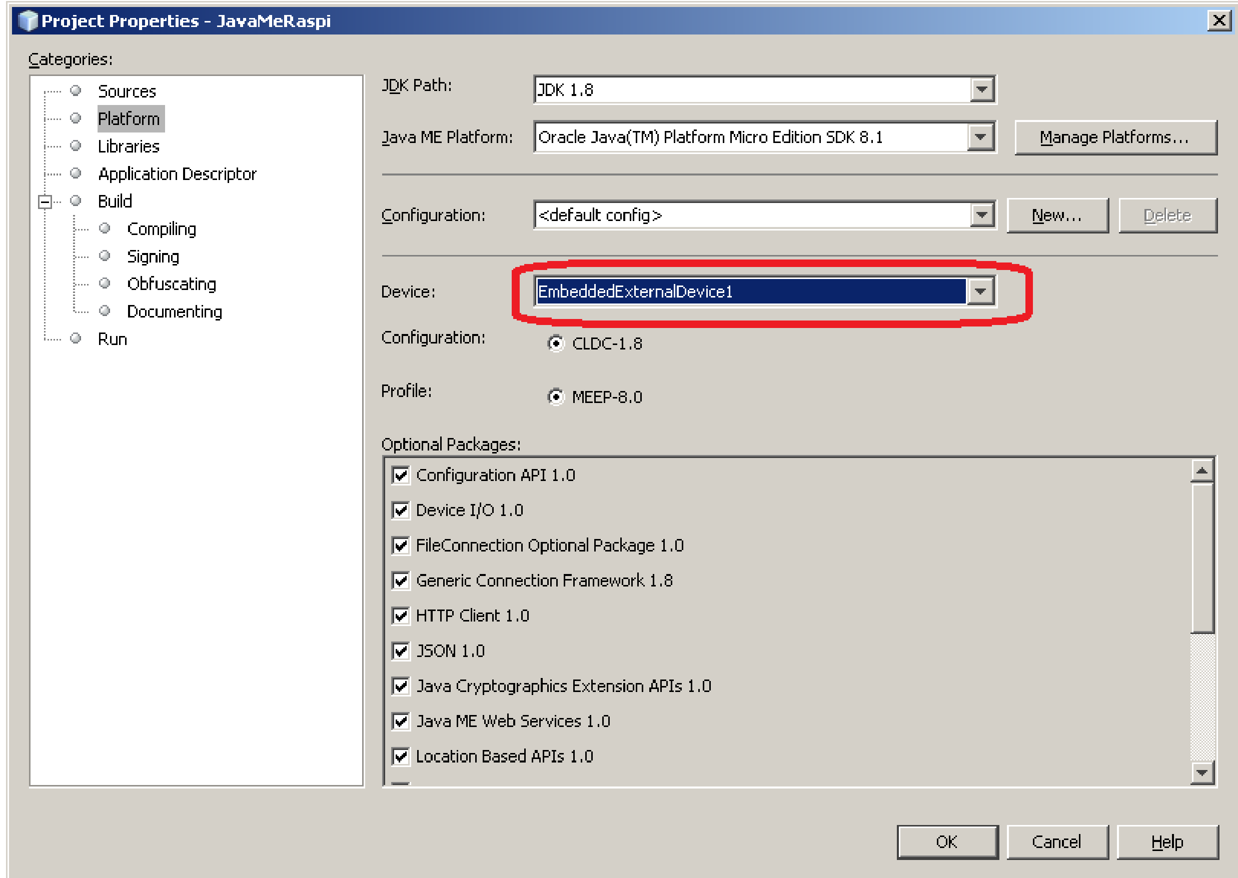
Not

R1 : Revision 1 R2 : Revision 2 Daha detaylı pin bilgileri için

Projeye Aygıtı Tanıtmak

Uygulama tamam, şimdi Device Manager'a tanıtılan Raspiyi Netbeans projesine tanıtmalıyız. Bunun için projeye sağ tuşlayarak `Properties` penceresine gidiyoruz ve `Platform` kısımından `EmbeddedExternalDeviceX` seçili olduğundan emin oluyoruz.

Şekil 14.15. EmbeddedExternalDeviceX'in Tanıtılması

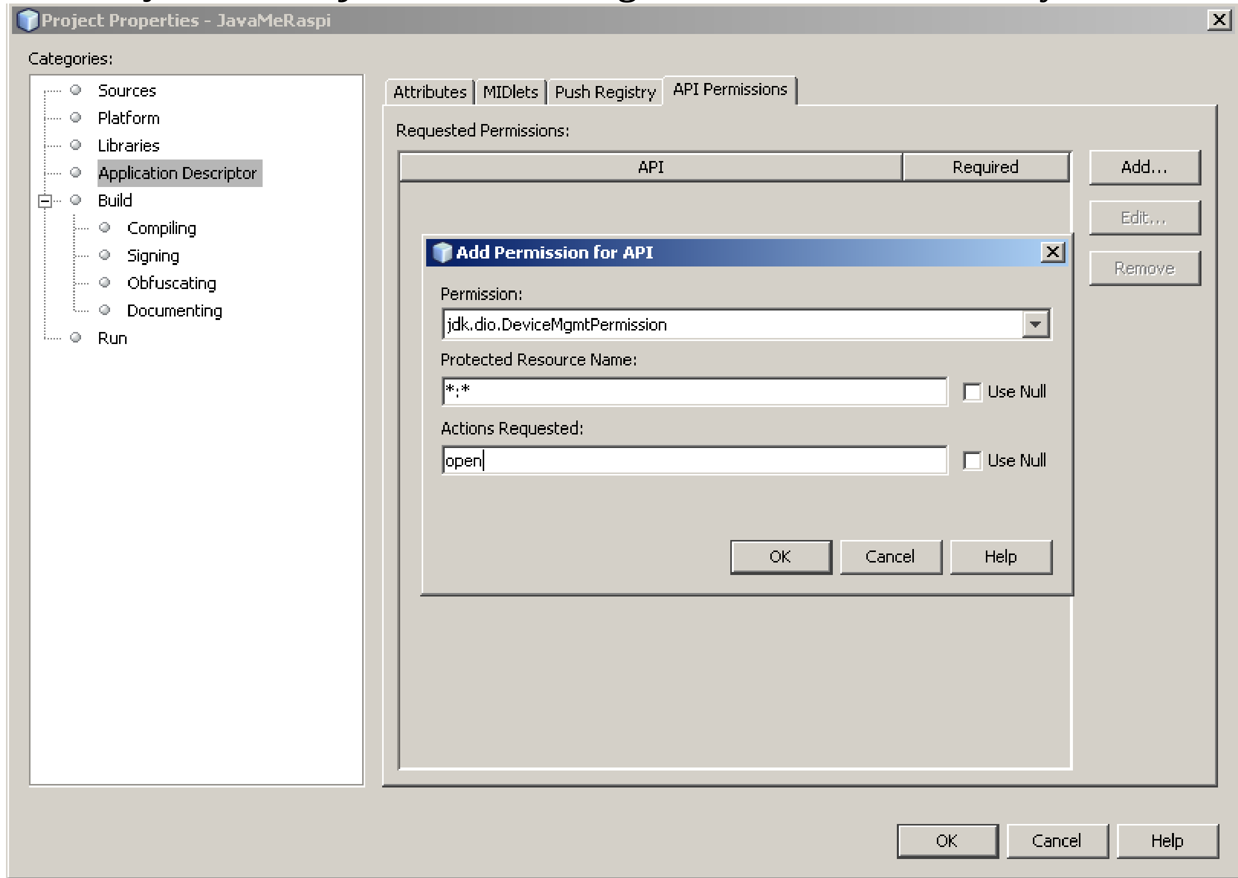


Uygulama İzinlerini Tanımlamak

Java ME uygulamaları gömülü sistem üzerinde çalışacakken, geliştiriciden sistemin hangi özelliklerini kullanmaya yetkin olacağına dair izinleri talep etmektedir. Bu sebeple uygulamamıza iki adet izin eklemeliyiz. Bunlar;

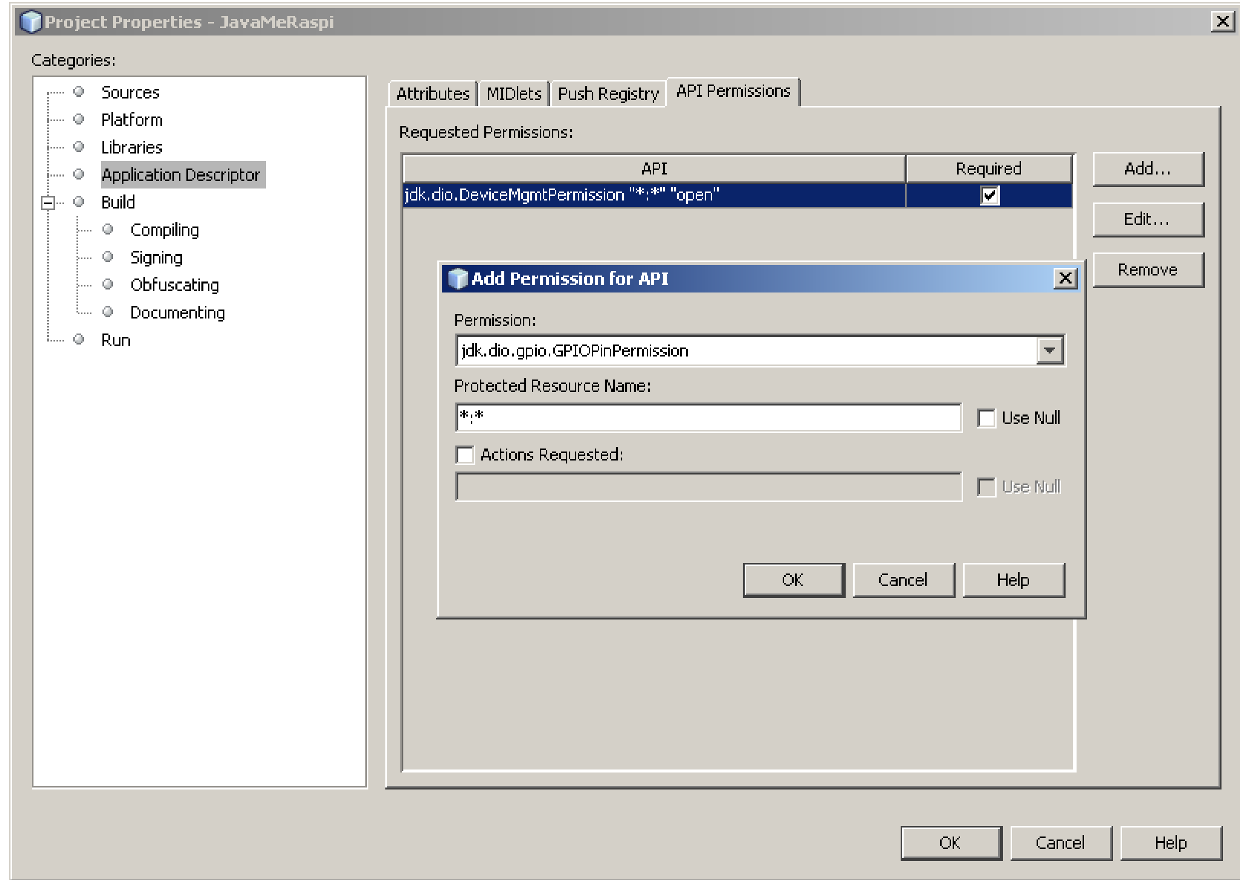
`jdk.dio.gpio.GPIOPinPermission` ve `jdk.dio.DeviceMgmtPermission` 'dir.

Şekil 14.16. jdk.dio.DeviceMgmtPermission izni veriliyor



Bu yetki cihazı yönetme yetkisi vermektedir.

Şekil 14.17. jdk.dio.gpio.GPIOPinPermission izni veriliyor



Bu yetki ise GPIO pinlerini kontrol etme yetkisi vermektedir.

Bu iki izin verildikten sonra tüm adımları tamamlamış oluyoruz.

Uygulamanın Çalıştırılması

Projeye sağ tuşlayarak `Run` dediğimizde, 5 numaralı porta bağlanan kırmızı LED'in birer saniye arayla yanıp söndüğünü görüyoruz.

Faydalanılan Kaynaklar

1. Java ME Embedded Getting Started Guide
2. RaspberryPi GPIO
3. wiringPi

Tekrar görüşmek dileğiyle..

Ek A. Bu kitap nasıl yazıldı?

Java 8 Ebook kitabı açık kaynak kodlu Asciidoc Fx kitap editörü kullanılarak, Asciidoc işaretleme dili ile yazılmıştır.